Design and Development of an Interactive Visualization Tools for Molecular Dynamics Simulation Data

Giacomo Garbin

29 January 2019

Supervised by

Pere-Pau Vázquez
MOVING Group - UPC

Marco Tarini
Department of Computer Science
University of Milan

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)
BarcelonaTech

Index

Abstract **3**

1 Introduction **4**
1.1 Motivation 5
1.2 Addressed Problem and Contribution 5
Root Mean Square Deviation 6
Root Mean Square Fluctuation 7
3D Visualization of RMSD and RMSF 7
1.3 Biomolecular Background 7
Our Case Study 9
1.4 Starting from Scratch 10

2 Space-Filling & Impostors **11**
2.1 Space-Filling 11
2.2 Real Sphere 12
Subdivision Process 12
Normalization Process 13
Sphere Mesh 15
2.3 Impostor Sphere 17
2.3.1 Ray Tracing 17
2.3.2 Impostor Generation 18
Vertex Shader 18
Geometry Shader 19
Fragment Shader 20
2.3.3 First View 22

3 Lighting **24**
3.1 Blinn-Phong Lighting Model 24
3.1.1 Light Components 25
Ambient Light 25
Diffuse Light 26
Specular Highlight 26
3.1.2 Turning on the Light 27
3.2 Screen-Space Ambient Occlusion (SSAO) 31
3.2.1 Geometry Step 32
3.2.2 Occlusion Step 33
3.2.3 Blur Step 38
3.2.4 Lighting Step 39

4 Object Outlining **42**
4.1 Silhouette & Outline 42
4.1.1 Silhouette Step 42

4.1.2 Outline Step 44
4.2 Multi-Outline for Atoms & Residues 46
4.2.1 Silhouette Step 47
4.2.2 Outline Step 48
4.3 Outline Color Mapping 50
4.3.1 Color Palettes 50
4.3.2 Color Scheme 51

5 RMSD & RMSF 54
5.1 Deviations & Fluctuations 54
5.2 Comparing Conformations 55
5.2.1 Proper Rigid Transformation 55
5.2.2 Optimal Alignment & Kabsch Algorithm 56
5.2.3 Minimum RMSD & RMSF 58
5.3 Code Implementation 59
5.3.1 Class Definitions 59
5.3.2 Method Definitions 61
5.3.3 Data Storage 66

6 User Interface 67
6.1 Central Area 67
6.1.1 Model Rotating 68
6.1.2 Model Panning 69
6.1.3 Camera Orbiting 70
6.1.4 Atom Selection 71
6.1.5 Camera Zooming 74
6.2 Right Panel 74
6.2.1 Atom and Residue Data Tab Widget 74
6.2.2 Trajectory Data Widget 75
6.2.3 Object Outlining Widget 75
6.2.4 Lighting Tab Widget 76
6.3 Bottom Bar 77
6.4 Shneiderman’s Mantra 77

7 Conclusions 78
7.1 Addressed Problem and Contribution 78
7.2 Accomplished Results 78
7.3 Application Improvements 79

Bibliography 81

Abstract

The research on molecular dynamics produces thousands of datasets pertaining the dynamic behavior of complex molecular compounds (including protein to protein docking, protein folding, and others), often by means of simulations, measurement. The work aims to be a contribution toward the identification of effective ways to visualize the outcome of molecular simulations.

This involves the design, development, and testing (including user studies) of interactive visualization tools, which conveys the inspected spatiotemporal data by means of dashboards, charts and 3D rendering of relevant protein-ligand configurations. The tools must be complete with a GUI and can potentially leverage clustering algorithms, rendering algorithms (implemented via OpenGL or WebGL), and display devices such as power-walls.

1 Introduction

Visualization is a discipline whose goal is to communicate information through images and it has a vast number of application areas. One of these areas is scientific visualization, which is focused on the visualization and inspection of scientific data in order to help scientists with understanding, knowledge discovery and hypothesis testing processes.

The work presented in this report belongs to the field of **molecular visualization**, a branch of scientific visualization that aims to visually represent molecules and their properties, and to help understand their behavior over time.

In many areas such as pharmacology or biotechnology, researchers use computers to perform **molecular simulations** (or MS) of the physical movements of atoms and molecules, and the interactions between molecules.

In the context of **drug design**, MS aim at predicting the binding of a small molecule, called **ligand** (the drug), with a larger biomolecule, the **protein**. Such a binding may inhibit or activate certain function of the biomolecule, which results in a therapeutic benefit for the patient.

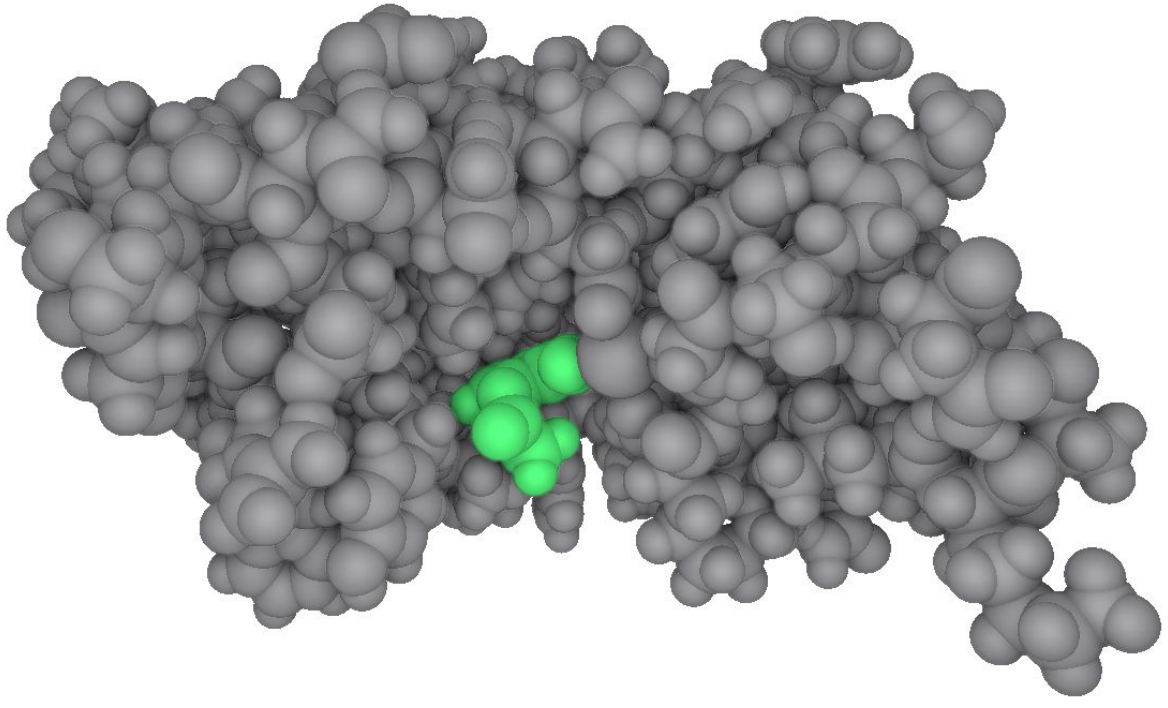


Figure 1.1 : docking process of a drug into a protein

The figure above shows the drug, highlighted in green, which attempts to enter an opening of the protein.

The result of a MS is a **trajectory** with the information of the positions of the atoms of both molecules in each step, together with associated data such as the temporal interactions that are established between the protein and the ligand and that guide the ligand towards the bound conformation, which they also help to stabilize.

1.1 Motivation

Computing molecular simulations (or MS) is a complex and time-consuming process, and analyzing the results generated can take days or weeks. Therefore providing new tools and techniques to visualize and interact with these simulations is crucial to understanding them.

The iterative loop in drug design typically involves simulations requiring long computation times, followed by a data analysis conducted by domain experts using also visualization tools. To help these experts during comprehension and decision making, it is of great importance to provide effective visualization tools.

Two of the key aspects required to make informed decisions in the drug design process are estimating the interaction energy between the protein and the ligand, and understanding which parts of the molecule influence their binding. This information enables the domain expert to hypothesize which residues can be altered in the subsequent design process in order to improve the ligand's affinity.

Communication of this information is a challenge.

1.2 Addressed Problem and Contribution

In contrast with the large efforts carried out in the field of molecular structural analysis, with a number of software packages providing all sorts of geometric information on the molecules, much less effort has been devoted to the development of efficient tools for the visual analysis of other variables such as the interaction forces.

In nowadays molecular visualization packages the methods available to visualize the interaction forces that drive these simulations are limited. Most of the software packages use simple 2D plots or a 2D visualization of the molecule of interest (the ligand) with the nearest interacting atoms, making difficult to understand the real 3D arrangement of these atoms.

Moreover, most of them have limited (or null) interaction and only allow the analysis of a single step of the simulation.

The following figure shows two examples of 2D visualization of the root mean square fluctuation (or RMSF).

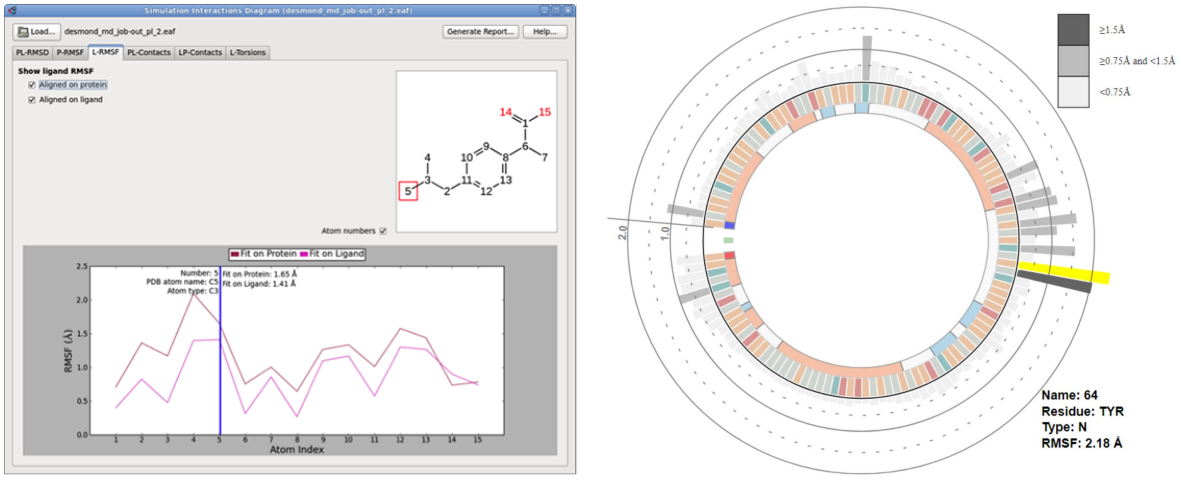


Figure 1.2 : two different 2D-methods to visualize RMSF (right side from [02])

The left side of the figure shows a screenshot of Schrödinger Maestro, where the RMSF is represented as a simple 2D plot. The right side of the figure shows the circular design proposed by P. Vázquez et al., which provides a link to the 3D structure of the molecule by maintaining the spatial order of the secondary structures, and, instead of visualizing this information as charts, they display the RMSF next to the protein backbone to facilitate relating the fluctuations with the residues involved. But still a 2D representation of the RMSF.

Our goal with the work presented in this report is to display directly on the 3D models of a molecular simulation the information useful in understanding the interactions between protein and ligand. In particular we will focus on two quantities: the **root mean square deviation** (or RMSD) and the **root mean square fluctuation** (or RMSF) of both atoms and residues.

Root Mean Square Deviation

The RMSD of atomic positions is the measure of the average distance between the atoms of superimposed molecules.

More generally, it is a measure of the average distance between two different conformations of a set of points in the three-dimensional space. In other words, the RMSD expresses how different two conformations are from each other.

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left\| x_{i,\,t} - x_{i,\,t_{ref}} \right\|^2}$$

When a dynamical system fluctuates about some well-defined reference position, the RMSD from the average over time can be referred to as the root mean square fluctuation (or RMSF).

Root Mean Square Fluctuation

The RMSF is a measure of the deviation of the position of a particle with respect to a reference position over time. Therefore this quantity can measure the average fluctuation of an atom around a reference position.

$$RMSF = \sqrt{\frac{1}{T}\sum_{t=1}^{T} \left\| x_{i,\,t} - x_{i,\,t_{ref}} \right\|^2}$$

The knowledge and visualization of both these quantities during an molecular simulation is a key element used by domain experts to detect important properties on the molecules involved and to better understand the interactions occurring during the simulation.

These are some examples.

- RMSD is used as a quantitative measure of similarity between two or more molecular structures.
- In the study of protein folding, RMSD is used as a reaction coordinate to quantify where the protein is between the folded state and the unfolded state.
- RMSD is commonly used in the context of docking to study the configuration of ligands when bound to macromolecules.
- RMSF is used to reveal movements both in the protein and the ligand derived from the (potential) energies. High values of this quantity may indicate that some parts of the protein move to interact with the ligand guiding it towards the binding site, to make room for the ligand, or to establish stabilizing interactions with the ligand.
- RMSF is used to reveal which are the most flexible parts of the molecule.

3D Visualization of RMSD and RMSF

The approach we proposed to visualize RMSD and RMSF directly on the molecule's 3D model is to **outline** the parts of the molecule involved into the study (which can be either individual atoms and residues) and map the values of RMSD and RMSF on a **color scheme**, which is then applied to the outline.

1.3 Biomolecular Background

A **molecule** is an electrically neutral group of two or more atoms held together by chemical bonds, while a **biomolecule** (or biological molecule) is a loosely used term for molecules that are present in organisms, essential to some typically biological process. The term biomolecules include also large macromolecules such as proteins.

In the following, we will mainly use the term molecule to refer to both concepts, it should be clear from the context to which we are actually referring.

Cells are made of molecules, they communicate through molecules, and they respond to signals by changing existing molecules or making new ones.

Molecules form the structures of cells and carry out cellular functions. Chemical bonds within and between biological molecules are important to their structure.

Most jobs in the cell are done by **proteins**, which form one of the groups of large molecules most important for cells.

To do their jobs, proteins must interact in a specific way with other molecules. The specific relationship between proteins and other molecules is based on the shape of the protein. Protein shape is complex and is essential to protein function.

Proteins are polymers[1] of **amino acids** (amino acids joined together by **peptide bonds**).

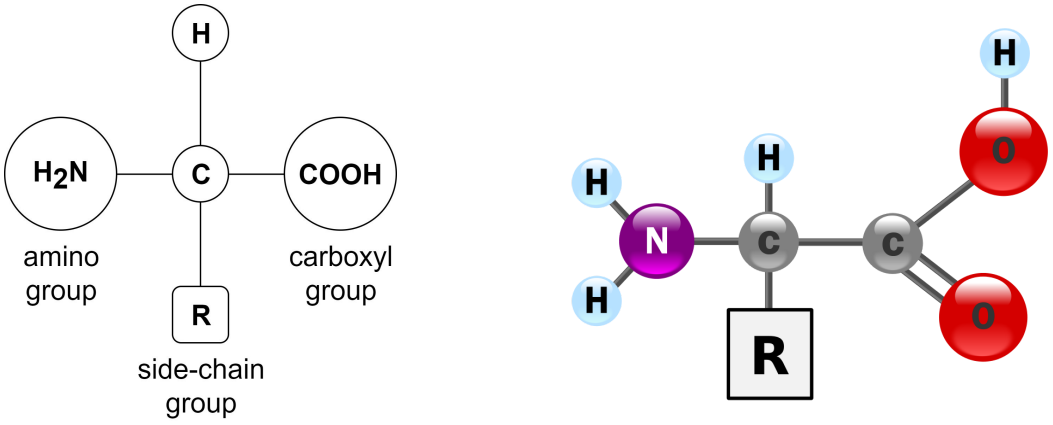


Figure 1.3 : amino acid structure (right side from Wikipedia)

The central carbon atom is flanked by an amino group and a carboxyl group.

The characteristic N-C-C sequence is called the **backbone** of the amino acids.

The name of the amino acid (or residue) depends on which one of the 20 side-chain groups is at R. The **side-chain** can be as simple as a hydrogen atom (as in the amino acid glycine) or more complicated.

A **residue** is a specific monomer within the protein, that is a molecule which can undergo polymerization, thereby contributing constitutional units to the essential structure of the macromolecule.

[1] *polymer* : a substance that has a molecular structure consisting chiefly or entirely of a large number of similar units bonded together

A polymer of amino acids is called a **polypeptide**. Once a polypeptide is folded and becomes functional, the polypeptide is called a protein. Some proteins are made of more than one folded polypeptide chain (quaternary structure).

Specific proteins are created based on the order of amino acids connected together and each polypeptide is folded in its native conformation (or native state), creating unique sections that are tailored for their particular job.

Protein shape is so important to their ability to do their job that if a protein unfolds (or denatures) it will no longer function, which can be the cause of diseases like Alzheimer's or Parkinson's diseases.

Protein structure is organized into four categories.

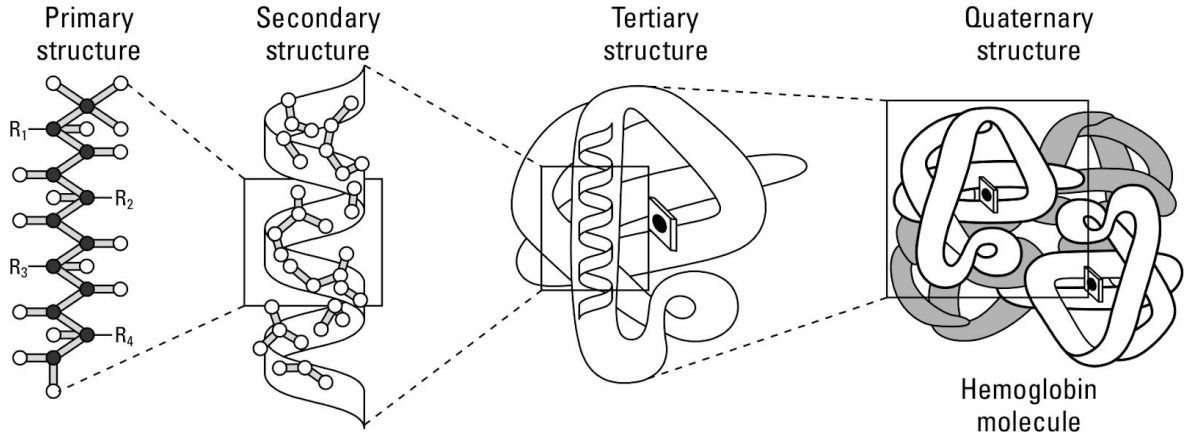


Figure 1.4 : protein structures (from [03])

- **primary structure** : is the sequence of amino acids (residues) in the protein
- **secondary structure** : are small areas that are folded into alpha helices and pleated sheets
- **tertiary structure** : is the final 3D shape of one folded amino acid chain
- **quaternary structure** : is found in proteins that consist of more than one folded amino acid chain

Our Case Study

In the course of the report we will test the features gradually added to the application on a particular case study: the interaction between an Aspirin (acetylsalicylic acid) and Phospholipase A2 (an enzyme).

The trajectory is composed of 234 steps or frames and illustrates the docking process of the Aspirin (the drug) into Phospholipase A2 (the protein), see Figure 1.1.

1.4 Starting from Scratch

In Section 1.2 we defined the objective of our work. It presuppose the creation of an application able to display three-dimensional models and which also allows the user to interact with them in real-time. In particular, the application can and must focus on the visualization of biomolecule models.

Based on these assumptions, we have decided to use the following technologies for the implementation of the application.

- **OpenGL** : (namely, Open Graphics Library) is an application programming interface (API) for rendering 3D graphics

- **Qt** : (pronounced “cute”) is a cross-platform application framework and widget toolkit for creating classic and embedded graphical user interfaces, and applications that run on various software and hardware platforms

Chapters 2 to 6 represent the core part of this document and illustrate the details of the application development.

- **Chapter 2** : this chapter discusses the representation method used to visualize the molecular model and how it is implemented in the application

- **Chapter 3** : this chapter illustrates the lighting model integrated into the application

- **Chapter 4** : this chapter shows how the outlining effect is implemented

- **Chapter 5** : this chapter discusses how the values of RMSD and RMSF have been calculated and how they are mapped to the outline

- **Chapter 6** : this chapter presents the user interface of the application and its features

2 Space-Filling & Impostors

In molecular visualization, various representation methods to visualize molecular models have been adopted. For our application we have chosen to adopt the space-filling model, which is one of the most commonly used methods.

This chapter describes the space-filling model and two different approaches to implement it. The first approach uses real spheres, while the second approach uses impostor spheres and it is de facto the standard nowadays in molecular visualization.

2.1 Space-Filling

The **space-filling** model is one of the most commonly used methods to visualize molecules, and it is probably also the most simplistic method, but nevertheless it gives a good approximation of the overall shape of the molecule.

This model represents each atom of the molecule using a sphere with its size determined by the element's **van der Waals radius**. Moreover, atoms of different chemical elements are usually represented by spheres of different colors.

The following table summarizes the values of the radii and colors attributed to the spheres in our application to differentiate the different chemical elements of the atoms which form the molecules of our case study.

element	radius in Å	radius in pm	albedo
Nitrogen	1.55	155	#3050f8
Carbon	1.70	170	#909090
Oxygen	1.52	152	#ff0d0d
Hydrogen	1.20	120	#ffffff
Sulfur	1.80	180	#ffff30
Calcium	2.31	231	#3dff00

Table 2.1 : space-filling radii and colors

The radius length of the spheres corresponds to the element's van der Waals radius (expressed both in angstroms and picometers[2]), while the color (or albedo) of the spheres uses the **CPK coloring**, a popular color convention for distinguishing atoms of different chemical elements in molecular models.

[2] $1\ Å = 10^{-10}\ m = 100\ pm$
$1\ pm = 10^{-12}\ m$

2.2 Real Sphere

There are several strategies to define a set of vertices (or points) arranged along the surface of a sphere. If the set is sufficiently large and the vertices it contains are well distributed along the surface, the surface of the sphere can be approximated with the triangulation process performed by the **OpenGL** rendering pipeline.

One of the possible strategies for defining a set of vertices arranged on a spherical surface is to use the vertices of an octahedron (see Figure 2.1) as a starting set and apply two processes, called the subdivision and normalization processes.

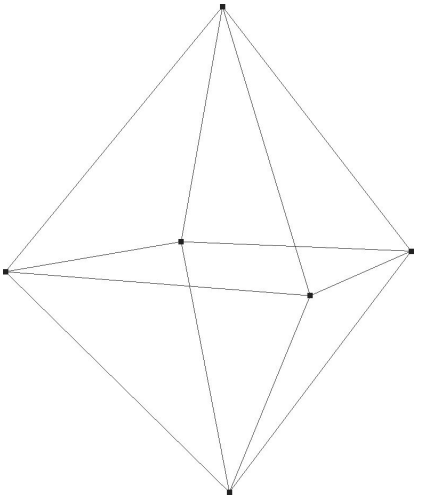
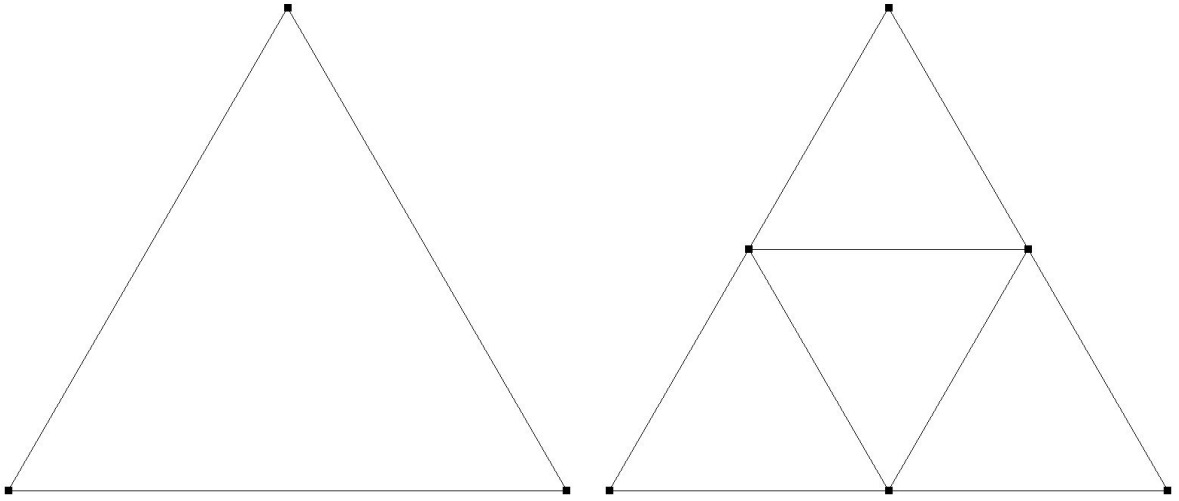


Figure 2.1 : octahedron mesh

Subdivision Process

The subdivision process consists of dividing each triangular face into four smaller triangular sub-faces. The process begins by considering the vertices of the triangle in pairs and calculating the midpoint of each pair of vertices. The three new vertices obtained, together with the three starting vertices, are used to define the four triangular sub-faces. Iterating the procedure, it is possible to obtain an arbitrary number of subdivision of the starting face.



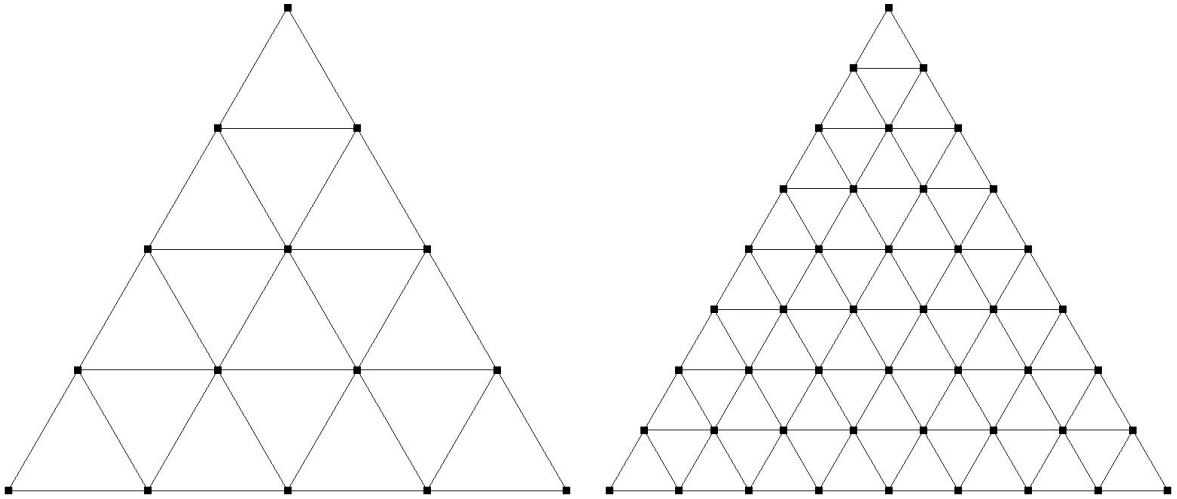


Figure 2.2 : three successive iterations of the subdivision process

The figure above illustrates three successive iterations of the subdivision process.

Normalization Process

The normalization process is applied to a set of vertices to modify their position so that they are all arranged at a given distance from a given point in space. This result can be easily obtained by imagining to trace the line connecting a vertex with the given point and then moving the vertex along this line, placing it at the desired distance.

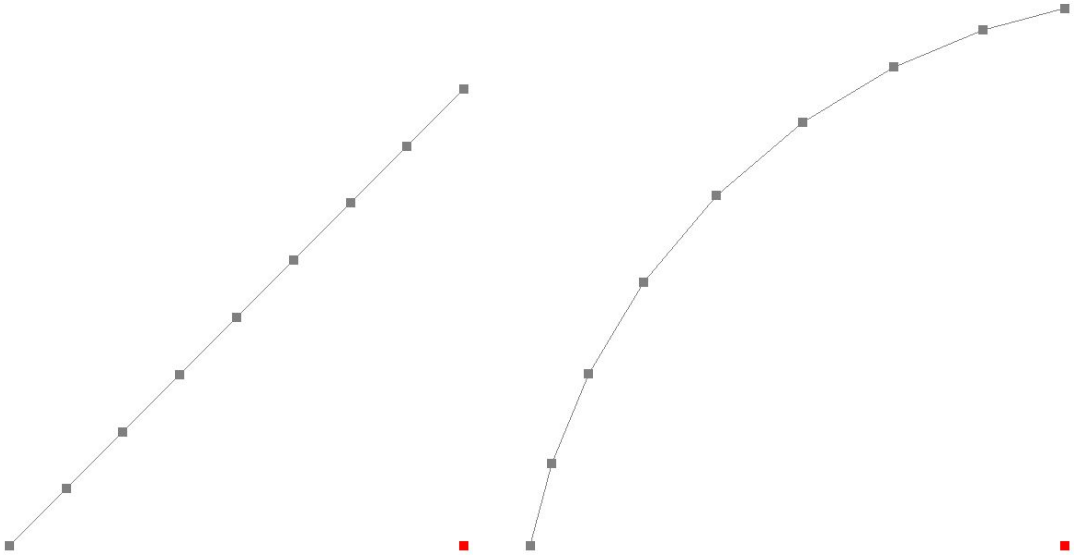
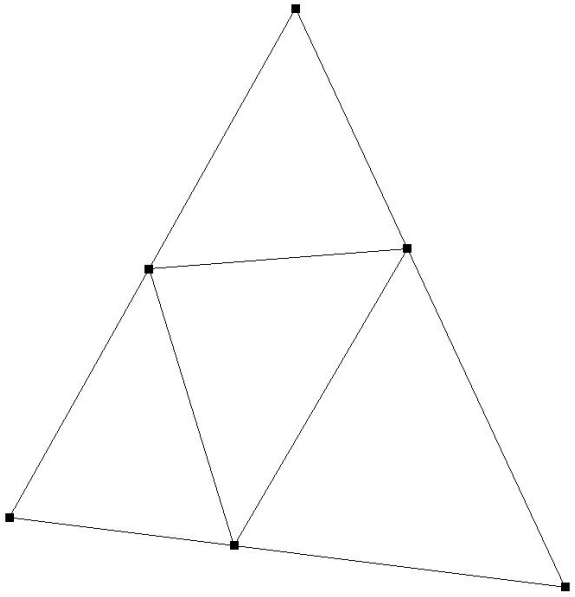
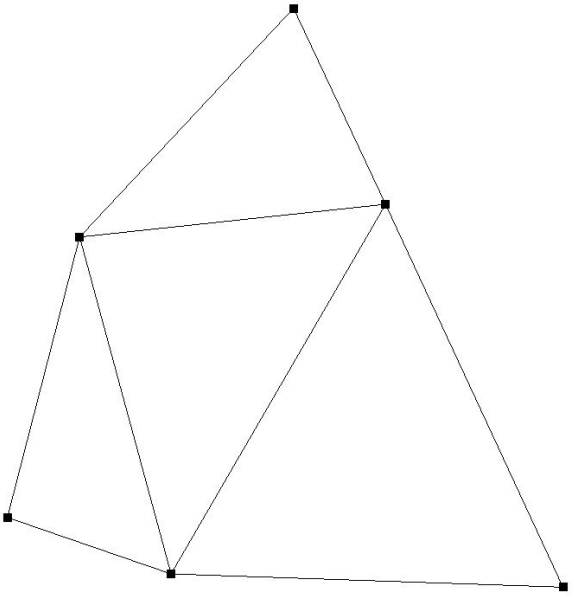
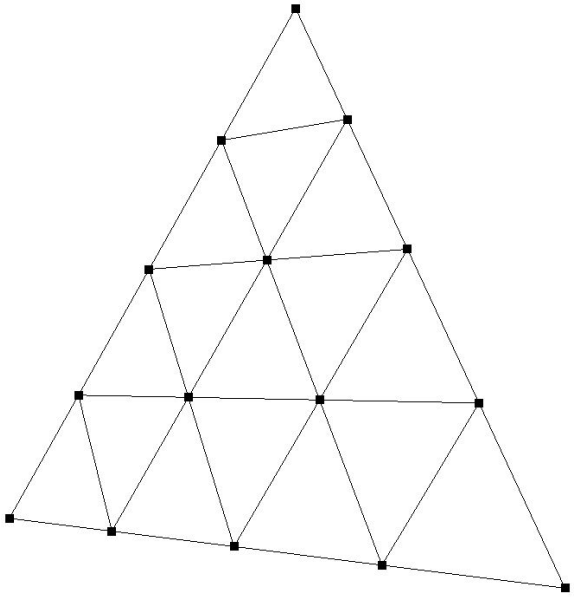


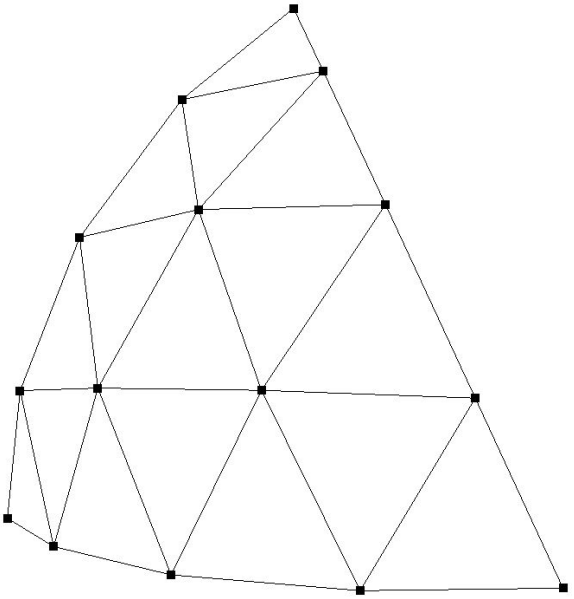
Figure 2.3 : a 2D example of the normalization process

The figure above illustrates a two-dimensional example of the normalization process. The points on the diagonal segment are normalized with respect to the point represented in red in the lower right corner. The normalization process is applied in the same way also in the three-dimensional space.









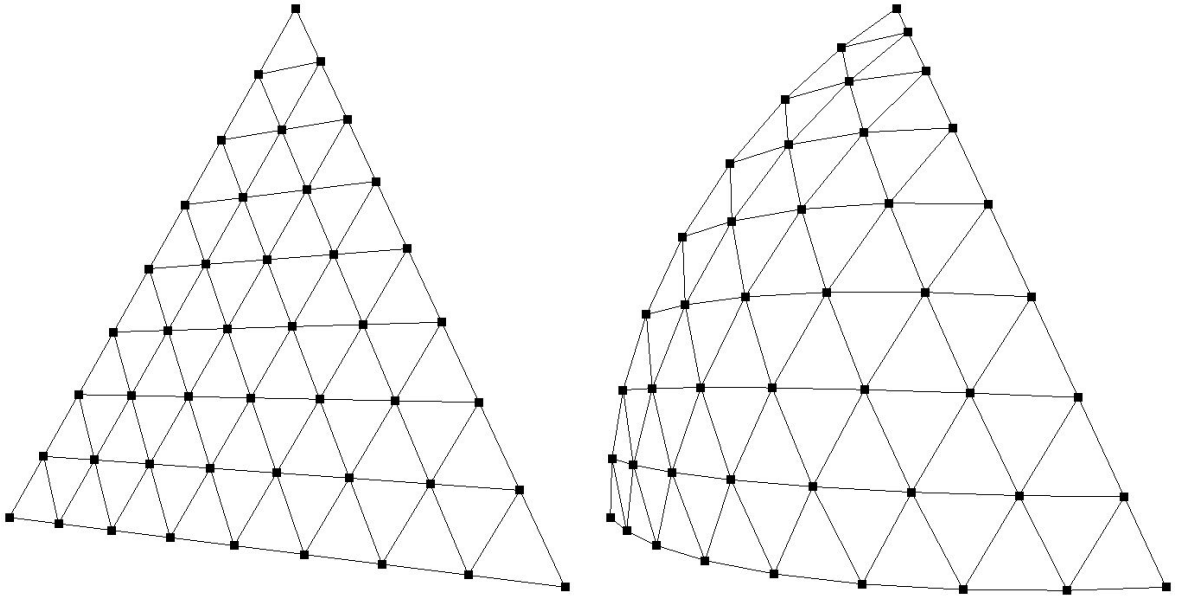


Figure 2.4 : a 3D example of the normalization process

The figure above illustrates a three-dimensional example of the normalization process applied to the vertices of a triangle, which has been subjected to different iterations of the subdivision process. The left side of the figure shows the vertices before being normalized while the right side of the figure shows the vertices after being normalized. The normalization process is made with respect to a point placed along the line parallel to the normal direction of the triangle face and passing through its center.

Sphere Mesh

Applying the subdivision and normalization processes to an octahedron is possible to obtain the approximation mesh of a sphere.

The following figure illustrates the normalization process applied to the vertices of an octahedron, whose triangular faces are subjected to successive iterations of the subdivision process.

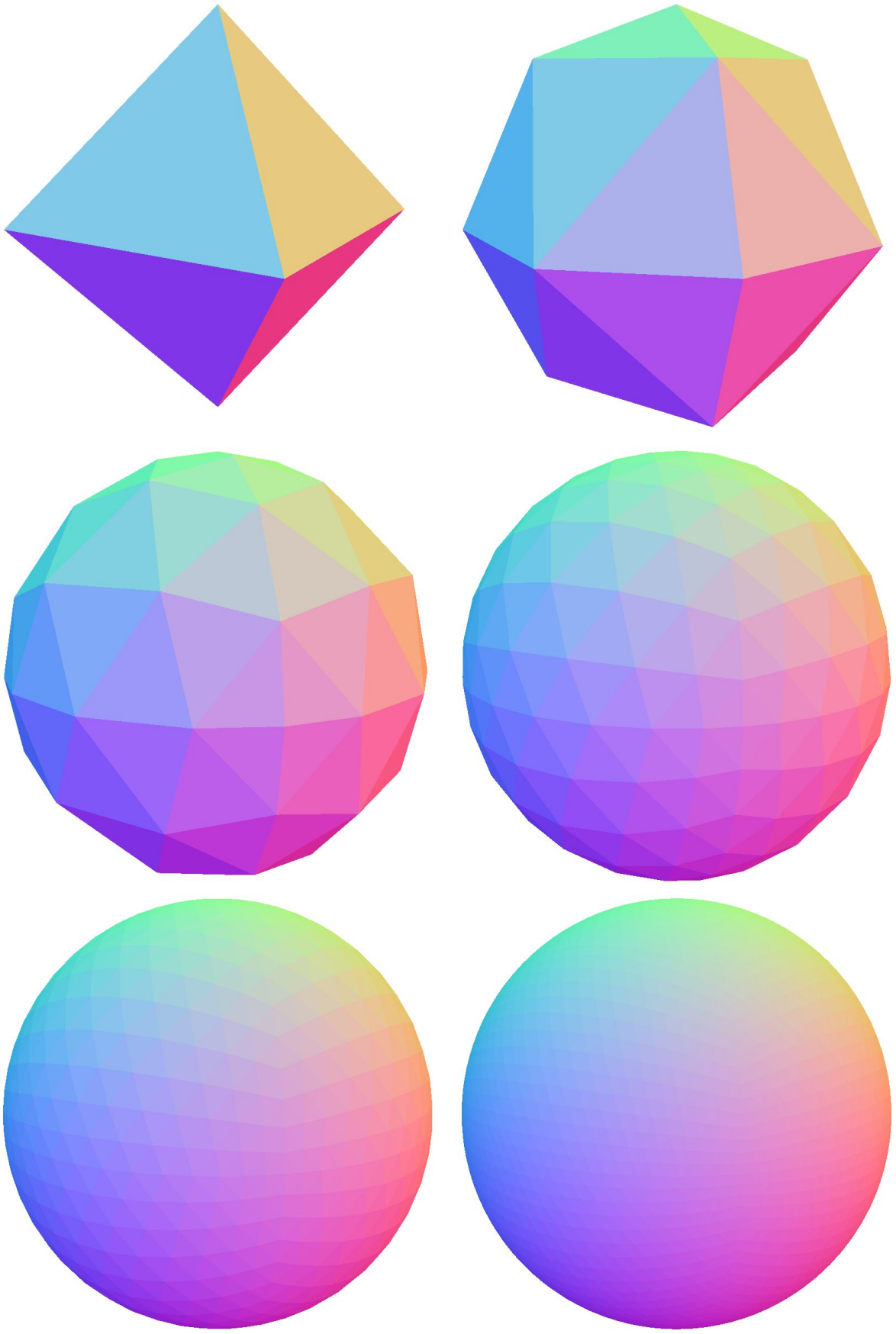


Figure 2.5 : subdivision and normalization processes applied to an octahedron

Iterating the subdivision process further, it is possible to obtain an arbitrary approximation of a sphere, but obviously the number of vertices of the mesh increases accordingly. Therefore, with this approach we must choose between performance and image quality, both defined by the number of triangles used to approximate the spheres.

2.3 Impostor Sphere

To overcome the problems inherent in the use of sphere meshes outlined in the previous section, it is de facto the standard nowadays in molecular visualization the use of impostors to render spheres.

An impostor sphere is a square or quad, in the jargon of the computer graphics called **billboard**, placed in the position where the sphere should be and which is always oriented towards the camera. In the fragment shader a circle is drawn procedurally on the billboard in order to map each point on the square to a point on the sphere we are trying to render.

There are two advantages compared to using sphere meshes. With impostors, only four vertices are needed to render each sphere and the spheres will always have the highest quality possible, limited only by the screen resolution.

2.3.1 Ray Tracing

In the fragment shader, we compute the position and normal of each point along the sphere's surface by using a technique called **ray tracing**. For each fragment, we create a ray from the camera position in the direction towards that point on the impostor square and we detect the point on the sphere that the ray hits, if there is any.

We can mathematically describe a ray with a pair of vectors, that is, the position $\overline{O}$ from which the ray is fired and the direction $\widehat{D}$ towards which the ray extends. The points on the ray can be expressed as the following equation.

$$\overline{P}(t) = \widehat{D}t + \overline{O}$$

Equation 2.1 : ray equation

The points on a sphere of center $\overline{S}$ and radius R can be expressed as the following equation.

$$\left\| \overline{P} - \overline{S} \right\| = R$$

We can substitute $\overline{P}$ with the ray equation. The ray is shot from the position of the camera, which in view-space is always $\overline{0}$, so $\overline{O} = \overline{0}$. With simple algebra, we obtain the following quadratic equation with respect to t.

$$\left(\hat{D}\cdot\hat{D}\right)t^2 - 2\left(\hat{D}\cdot\overline{S}\right)t + \left(\overline{S}\cdot\overline{S}\right) - R^2 = 0$$

If the discriminant is negative, then the equation has no solution. In terms of our ray test, this means the ray misses the sphere.
If the discriminant is positive, then the equation has two solution. The ray hits the sphere in two places: once going in, and once coming out. We're interested in the smallest solution, the one in front of the camera.

Once we find a solution for t, we can use the ray equation to compute the position. With the position and the center of the sphere, we can also compute the normal.

2.3.2 Impostor Generation

The generation of each impostor sphere follows the steps shown in Figure 2.6.

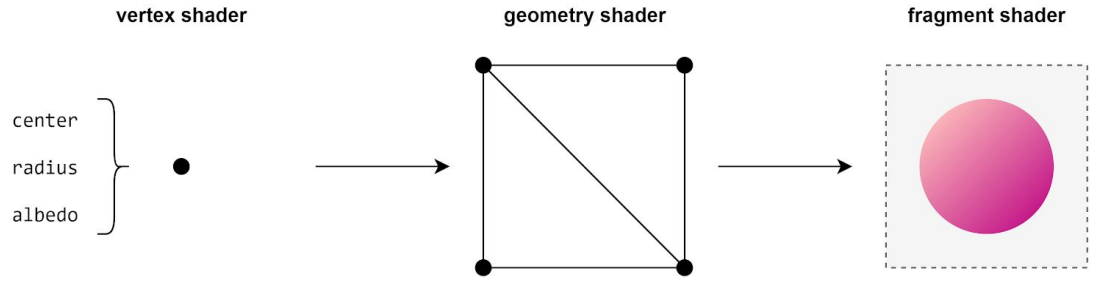


Figure 2.6 : overview on impostor sphere generation

The vertex shader receives the data of the sphere, the geometry shader builds the quad and the fragment shader calculates the position and normal of each point of the sphere.

Vertex Shader

The vertex shader is invoked only once for each sphere and generates a single point.

Listing 2.1 : impostor vertex shader

```
#version 450 core

layout (location = 0) in vec3 center;
layout (location = 1) in float radius;
layout (location = 2) in vec3 albedo;

out Vertex
{
  vec3 center;
  float radius;
  vec3 albedo;
} vert;

void main()
```

```
{
  vert.center = center;
  vert.radius = radius;
  vert.albedo = albedo;
}
```

The vertex shader is straightforward. It receives the information of the sphere as input, namely the position of the center, the radius and the color of the sphere. In our case, each sphere represents an atom of a molecule and the values of radius and albedo will be those that distinguish the chemical element of the atom.

Geometry Shader

The geometry shader receives the point generated by the vertex shader as input. For each received point, it generates four new vertices, each deviated so as to form the vertices of the billboard, which is returned as a triangle strip.

Listing 2.2 : impostor geometry shader

```
#version 450 core

layout (points) in;
layout (triangle_strip, max_vertices = 4) out;

in Vertex
{
  vec3 center;
  float radius;
  vec3 albedo;
} vert[];

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out Fragment
{
  flat vec3 center;
  flat float radius;
  flat vec3 albedo;
  smooth vec2 offset;
} frag;

void main()
{
  vec2 offsets[4];
  offsets[0] = vec2(-1.0f, -1.0f);
  offsets[1] = vec2(-1.0f, +1.0f);
  offsets[2] = vec2(+1.0f, -1.0f);
  offsets[3] = vec2(+1.0f, +1.0f);

  float BoxCorrection = 1.5f;

  for (int i = 0; i < 4; i++)
```

```
  {
    gl_Position = view * model * vec4(vert[0].center, 1.0f);

    frag.center = vec3(gl_Position);
    frag.radius = vert[0].radius;
    frag.albedo = vert[0].albedo;
    frag.offset = offsets[i] * BoxCorrection;

    gl_Position.xy += (frag.offset * frag.radius);
    gl_Position = projection * gl_Position;

    EmitVertex();
  }

  EndPrimitive();
}
```

The geometry shader also expands the size of the square, in order to avoid the problem illustrated in Figure 2.7.

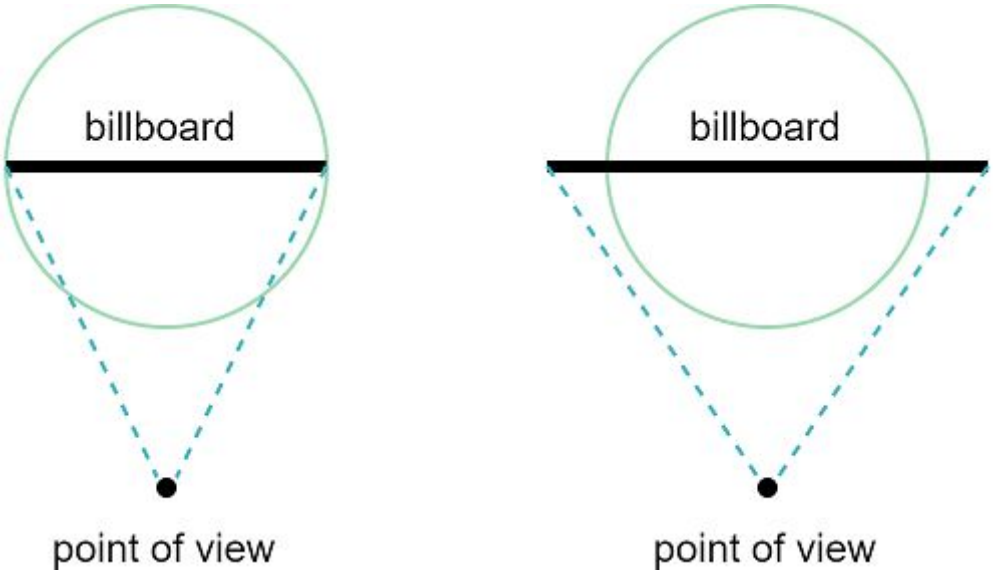


Figure 2.7 : billboard size correction

If the square were the same size as the sphere, in certain configurations, such as with large spheres or after a large zoom-in, some parts of the impostor would be cut off.

Fragment Shader

The fragment shader applies the ray tracing algorithm described in Section 2.3.1 and returns the position, normal vector and color of each pixel in three different color buffers. Note that the exported positions and normals are in view-space.

Listing 2.3 : impostor fragment shader

```
#version 450

layout (location = 0) out vec3 center;
layout (location = 1) out vec3 normal;
layout (location = 2) out vec3 albedo;

struct Impostor
{
  vec3 center;
  vec3 normal;
};

void SetImpostor(out Impostor impostor);

in Fragment
{
  flat vec3 center;
  flat float radius;
  flat vec3 albedo;
  smooth vec2 offset;
} frag;

uniform mat4 projection;

void main()
{
  Impostor impostor;
  SetImpostor(impostor);

  // frag depth
  vec4 clip = projection * vec4(impostor.center, 1.0f);
  float depth = clip.z / clip.w;

  float near = gl_DepthRange.near;
  float far = gl_DepthRange.far;
  float diff = gl_DepthRange.diff;
  gl_FragDepth = ((diff * depth) + near + far) * 0.5f;

  center = impostor.center;
  normal = impostor.normal;
  albedo = frag.albedo;
}

void SetImpostor(out Impostor impostor)
{
  vec3 ray = normalize(frag.center + vec3(frag.offset * frag.radius, 0.0f));

  float b = 2.0f * dot(ray, -frag.center);
  float c = dot(frag.center, frag.center) - (frag.radius * frag.radius);

  float delta = (b * b) - (4.0f * c);

  if(delta < 0.0f)
  {
    discard;
```

```
  }

  delta = sqrt(delta);
  float t1 = (-b + delta) * 0.5f;
  float t2 = (-b - delta) * 0.5f;
  float t = min(t1, t2);

  impostor.center = ray * t;
  impostor.normal = normalize(impostor.center - frag.center);
}
```

The impostor sphere has no depth, it's just a circle on the billboard. We have to write the depth value ourselves going through the process OpenGL normally goes through to compute the depth. The view-space position is transformed to clip-space through the projection matrix. The perspective division happens, transforming to normalized device coordinate (NDC) space. The depth range function is applied, forcing the [-1, +1] range in the fragment shader to the range provided with glDepthRange. We write the final depth to a built-in output variable gl_FragDepth.

2.3.3 First View

We can finally take a look to the model of our case study.

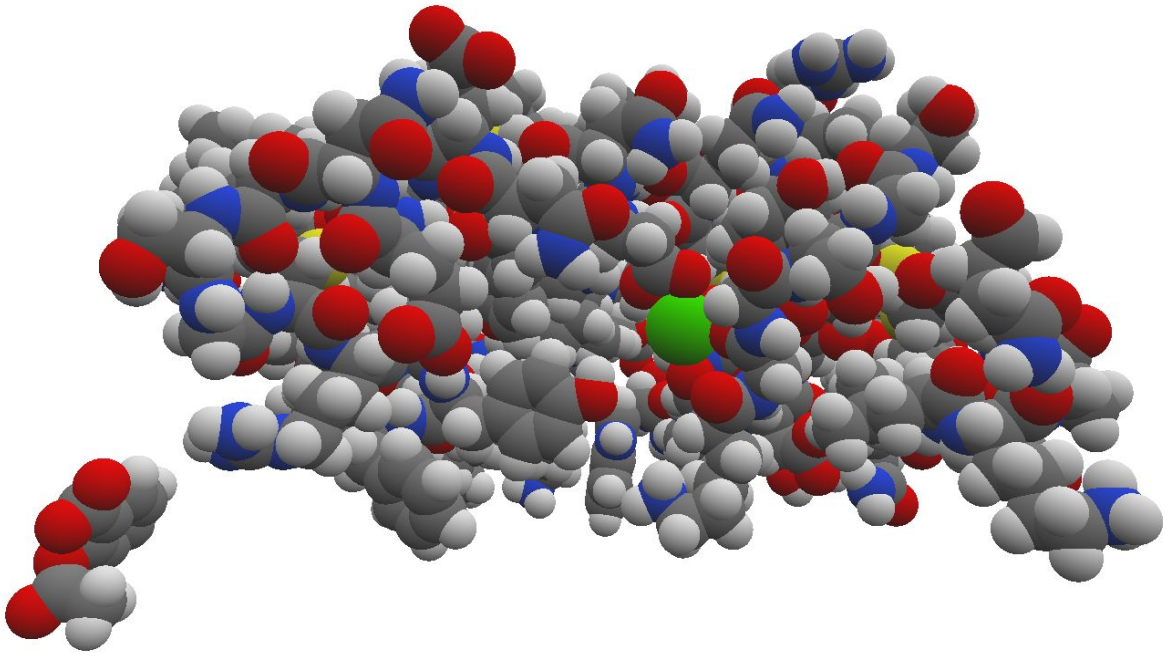


Figure 2.8 : a first look at the model of our case study

In the figure above, each fragment of the model is assigned the corresponding albedo value attenuated with a grayscale factor based on the normal vector of the fragment.

Although the rendered image in Figure 2.8 allows us to perceive the atoms configuration of the molecules, does not return a great feeling of realism. In the next chapter we will use the information generated by the sphere impostor fragment shader to improve the realism of the image by integrating a lighting model into the project.

3 Lighting

In the previous chapter we illustrated an approach, called impostors, which allows us to visualize a scene with a large quantity of spherical models (which will represent the atoms of our molecules), rendered with the highest quality possible and with a small computational cost.

Moreover, the impostor's strategy returns for each sphere two precious geometrical information, namely the position and normal vectors of each fragment, which we will use in this chapter for lighting calculations.

At the end of the previous chapter we tried to render the model of our case study using only the information available to us, without further calculations (see Figure 2.8). Although the resulted image allows us to perceive the atoms configuration of the molecules, does not return a great feeling of realism.

In this chapter we will integrate a lighting model into the application, improving the realism transmitted by the generated images, which in turn facilitates the understanding of the configuration of molecular models.

3.1 Blinn-Phong Lighting Model

The simulation of light in computer graphics is a huge topic.

Many approaches have been proposed, some of them very complex and expensive in terms of computational resources, and therefore not suitable for real-time applications, while other models try to find a good compromise between the realism of the simulation and its performance.

To this second category belongs also the lighting model that we will integrate in the project, which is known as the **blinn-phong** lighting model.

The process of lighting in our application consists of two phases or steps.

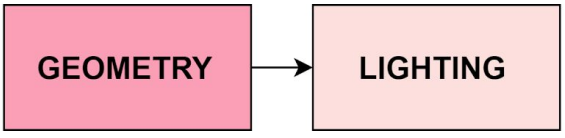


Figure 3.1 : steps of lighting process

- **Geometry Step** : in this step the geometry of the scene is calculated and then "exported" to be used in the next stage

- **Lighting Step** : in this step the lighting model is applied and the resulting image is return to the user

This two-step partition is known as deferred rendering and, although not strictly necessary to apply the blinn-phong model, we will exploit it in the next section, where we will add the screen-space ambient occlusion to our lighting model.

3.1.1 Light Components

The blinn-phong model divides the lighting into three components.

- **ambient light** : is an approximation of the scattered light present in a scene that does not come from any specific light source or direction

- **diffuse light** : is the light scattered by a surface equally in all directions due to a particular light source

- **specular highlight** : is the light reflected directly by the surface

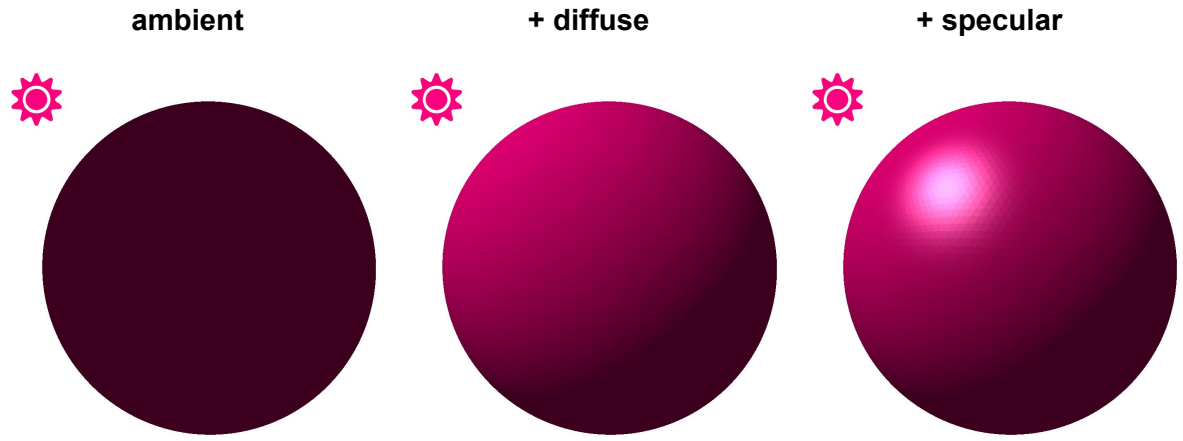


Figure 3.2 : light components in blinn-phong model

The figure above shows the application of the three components of the blinn-phong lighting model to the mesh of a real sphere constructed with the approach described in section 2.2. On the left only the ambient component is applied, in the middle the diffused component is added and on the right the specular component is also added.

Ambient Light

This light component does not come from any specific light source or direction and is an approximation of the scattered light present in a scene.

The blinn-phong lighting model considers it a **constant** throughout the scene.

In the next section we will dynamically compute this component based on the geometry of the scene with a technique called **screen-space ambient occlusion**.

Diffuse Light

This light component is the light scattered by a surface equally in all directions due to a particular light source.

It doesn't matter which direction the eye is, but it does matter which direction the light is. It is brighter when the surface is more directly facing the light source, simply because that orientation collects more light than an oblique orientation.

Diffuse light computation depends on the direction of the surface normal and the direction of the light source, but not the direction of the eye. It also depends on the color of the surface.

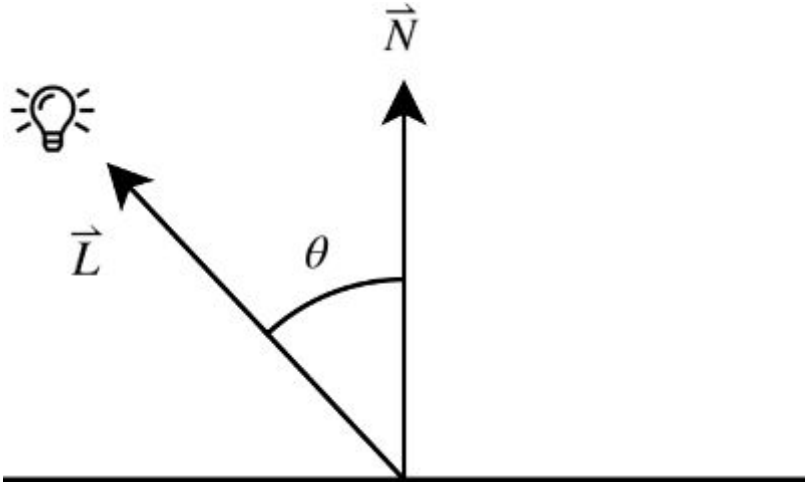


Figure 3.3 : diffuse light computation

For each fragment, we calculate the diffuse factor as the cosine of the angle θ between the light direction $\vec{L}$, which is the vector given by the difference between the light and fragment positions, and the fragment normal $\vec{N}$.

If both $\vec{L}$ and $\vec{N}$ are normalized, the cosine of θ is simply the dot product of the two vectors. The diffuse factor will be maximum when $\vec{L}$ and $\vec{N}$ are aligned (i.e., when θ is zero) and we force it to zero when θ is greater than 90 degrees, which means that the light is below the surface.

Specular Highlight

This light component is the light reflected directly by the surface.

It refers to how much the surface material acts like a mirror. A metal-like surface reflects a very sharp bright specular highlight, while a duller surface reflects a larger, dimmer specular highlight. The strength of this angle-specific effect is referred to as **shininess**. The higher

the shininess value of a surface, the more it sharply reflects the light instead of scattering it all around.

Computing specular highlights requires knowing how close the surface's orientation is to the needed direct reflection between the light source and the eye. Hence, it requires knowing the surface normal, the direction of the light source, and the direction of the eye.

Specular highlights might or might not incorporate the color of the surface. Typically, it is more realistic to not involve any surface color, making it purely reflective. The underlying color will be present anyway from the diffuse term, giving it the proper tinge.

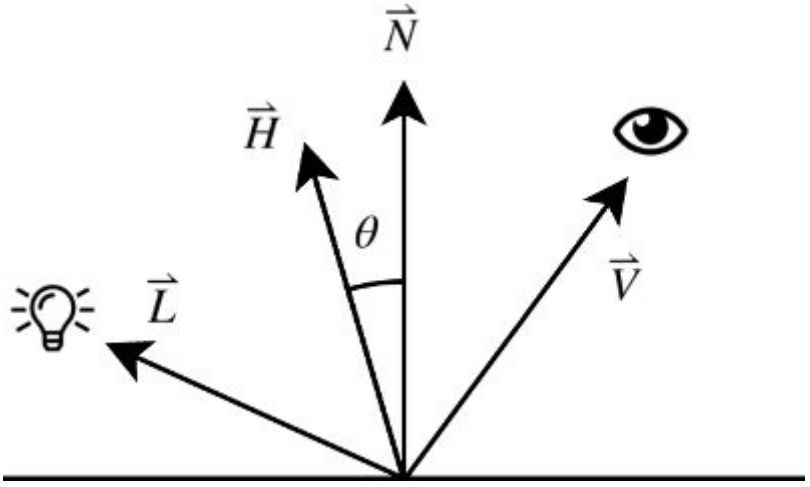


Figure 3.4 : specular highlight computation

For each fragment, we first compute the angle θ between the halfway direction $\vec{H}$ and the fragment normal $\vec{N}$. $\vec{H}$ is a unit vector halfway between the view direction $\vec{V}$, which is the vector given by the difference between the eye and the fragment positions, the light direction $\vec{L}$ (as before, the vector given by the difference between the light and fragment positions). Then we calculate the specular factor by raising the cosine of θ to the power of the shininess value of the highlight.

If both $\vec{H}$ and $\vec{N}$ are normalized, the cosine of θ is simply the dot product of the two vectors. The specular factor will be maximum when $\vec{H}$ and $\vec{N}$ are aligned (i.e., when θ is zero) and we force it to zero when θ is greater than 90 degrees.

In the original phong lighting model, θ is calculated as the angle between the fragment normal $\vec{N}$ and the light reflection direction around $\vec{N}$, rather than the halfway direction $\vec{H}$.

3.1.2 Turning on the Light

We pass the following information to the lighting fragment shader.

- **center and normal textures** : the geometry of the scene

- **albedo texture** : the colors of the scene
- **shininess value** : for specular highlight
- **light position and components**

The textures center, normal and albedo are those returned by the geometry step (see Listing 2.3).

Listing 3.1 : lighting fragment shader

```
#version 450 core

struct Fragment
{
  vec3 center;
  vec3 normal;
  vec3 albedo;
};

struct Light
{
  // light position in view-space
  vec3 center;
  // light components
  vec3 ambient;
  vec3 diffuse;
  vec3 specular;
};

void SetFragment(out Fragment frag);
vec3 GetLighting(Fragment frag, Light light, vec3 eye);

smooth in vec2 offset;

uniform sampler2D center;
uniform sampler2D normal;
uniform sampler2D albedo;

uniform float shininess;
uniform Light light;

out vec4 PixelColor;

void main()
{
  Fragment frag;
  SetFragment(frag);

  if (frag.albedo == vec3(0.0f))
  {
    discard;
  }

  // in view-space the eye position is in the origin of the coordinate system
  vec3 eye = vec3(0.0f);
```

```
  vec3 lighting = GetLighting(frag, light, eye);
  PixelColor = vec4(lighting, 1.0f);
}

void SetFragment(out Fragment frag)
{
  frag.center = texture(center, offset).xyz;
  frag.normal = texture(normal, offset).xyz;
  frag.albedo = texture(albedo, offset).rgb;
}

vec3 GetLighting(Fragment frag, Light light, vec3 eye)
{
  float factor;

  // ambient component
  vec3 ambient = light.ambient;

  // diffuse component
  vec3 LightDirection = normalize(light.center - frag.center);
  factor = max(dot(frag.normal, LightDirection), 0.0f);
  vec3 diffuse = factor * light.diffuse;

  // specular component
  vec3 EyeDirection = normalize(eye - frag.center);
  vec3 halfway = normalize(LightDirection + EyeDirection);
  factor = pow(max(dot(frag.normal, halfway), 0.0f), shininess);
  vec3 specular = factor * light.specular;

  vec3 lighting = (ambient + diffuse + specular) * frag.albedo;
  return lighting;
}
```

For each fragment, we first retrieve the geometry and colors of the scene and then assign the actual color due to the lighting computation to the fragment.

The GetLighting() method implements the theory discussed in the previous section. It calculates the three components of the blinn-phong model, adds them together and multiplies the result by the fragment albedo to obtain its actual color.

All coordinates are assumed to be in view-space, therefore the eye position is in the origin of the coordinate system.

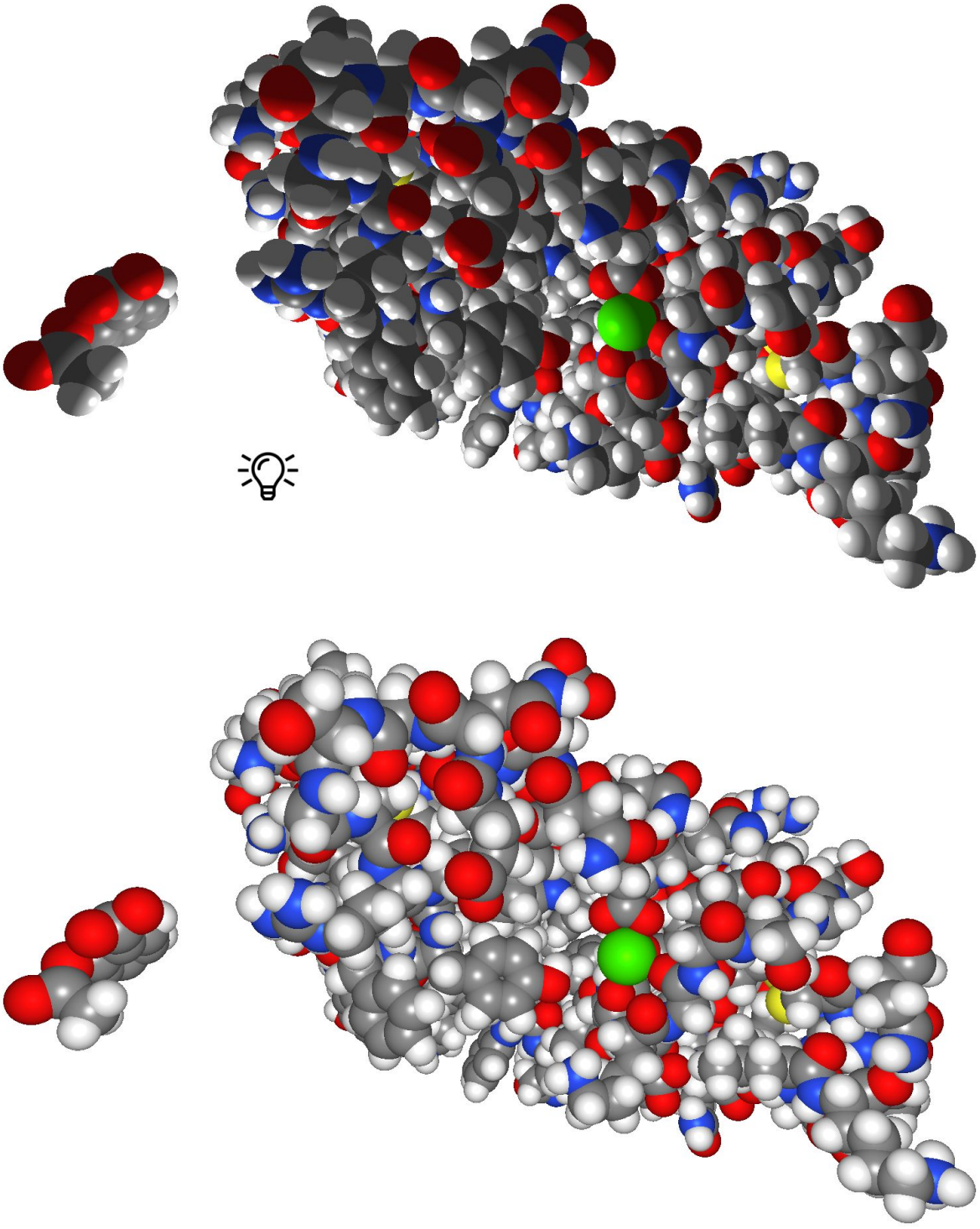


Figure 3.5 : blinn-phong lighting model applied to our case study

The figure above shows the blinn-phong lighting model applied to our case study. At the top of the figure, a white light is positioned next to the model. The three components of the blinn-phong model can be clearly recognized. Observe how the parts not directly reached by the light are darker. Observe also how the light components, in particular the specular highlight, change based on the albedo of each sphere. At the bottom of the figure, the same scene is rendered with the same white light, but this time positioned in the camera position.

The introduction of a lighting model greatly increases the sense of realism perceivable in the images generated by the application, compare Figure 3.5 with Figure 2.8 of the previous chapter.

However, when a sphere is illuminated, the current model does not take into account the neighboring spheres. All objects receive the same lighting, based exclusively on the direction between light and object, regardless of the surrounding environment.

This is the issue we will tackle in the next section with the use of a technique for the dynamic computation of the ambient light component.

3.2 Screen-Space Ambient Occlusion (SSAO)

In the Blinn-Phong lighting model, illustrated in the previous section, the ambient component is simply a constant. In reality, it depends on the geometry of the scene to which the lighting model is applied. A widespread technique that approximates indirect light in real-time is called **screen-space ambient occlusion** (or SSAO).

SSAO generates an **occlusion factor** for each fragment based on the depth value of the surrounding fragments, which is then used to attenuate the ambient component of the lighting model.

As the name suggests, SSAO works in screen-space, and is part of a family of techniques, called **deferred rendering**, in which the lighting calculation is separated by and postponed to (deferred to) the calculation of the scene geometry.

SSAO consists of four phases or steps.

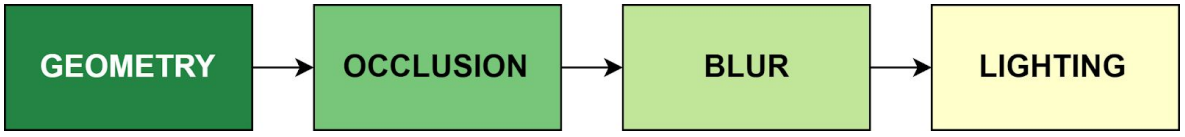


Figure 3.6 : steps of screen-space ambient occlusion

- **Geometry Step** : in this step the geometry of the scene is calculated and then "exported" to be used in the following stages
- **Occlusion Step** : in this step the occlusion factor of each fragment is calculated
- **Blur Step** : in this step a blur effect is applied to the result of the previous stage to obtain a smooth occlusion factor
- **Lighting Step** : in this step the lighting model is applied and its ambient component is attenuated using the occlusion factor generated in the previous stages

3.2.1 Geometry Step

SSAO calculates the occlusion factor in view-space. In this step the scene geometry, that is, positions and normals of the vertices, are transformed into view-space and stored in two textures: center and normal. This fits well with the approach used to generate the impostor spheres described in the previous chapter, since it also works in view-space. Moreover, another texture called albedo is filled with the colors of the vertices.

Listing 3.2 : geometry fragment shader

```
#version 450

layout (location = 0) out vec3 center;
layout (location = 1) out vec3 normal;
layout (location = 2) out vec3 albedo;

struct Impostor
{
  vec3 center;
  vec3 normal;
};

void SetImpostor(out Impostor impostor);

in Fragment
{
  flat vec3 center;
  flat float radius;
  flat vec3 albedo;
  smooth vec2 offset;
} frag;

uniform mat4 projection;

void main()
{
  Impostor impostor;
  SetImpostor(impostor);

  // ...

  center = impostor.center;
  normal = impostor.normal;
  albedo = frag.albedo;
}
```

The fragment shader of the geometry step calculates the scene geometry and exports the results in three different color buffers, attached respectively to the center, normal and albedo textures.

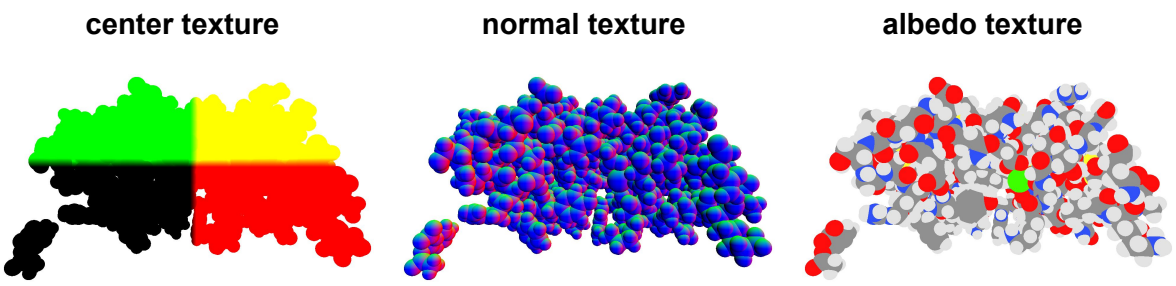


Figure 3.7 : result of the geometry step applied to our case study

The figure above shows the result of the geometry step applied to our case study.

3.2.2 Occlusion Step

SSAO generates an occlusion factor for each fragment based on the depth value of the surrounding fragments. The samples that make up the sample kernel are distributed within a hemisphere tangent to the fragment, with the normal vector pointing in the positive z direction.

Listing 3.3 : SSAO kernel definition

```
void SetKernel()
{
  QRandomGenerator generator = QRandomGenerator::securelySeeded();

  kernel.clear();

  auto lerp = [] (float a, float b, float f)
  {
    return a + f * (b - a);
  };

  float size = KernelSize * KernelSize;

  for (int i = 0; i < size; i++)
  {
    QVector3D sample;
    sample.setX(generator.generateDouble() * 2.0f - 1.0f);
    sample.setY(generator.generateDouble() * 2.0f - 1.0f);
    sample.setZ(generator.generateDouble());
    sample.normalize();
    sample *= generator.generateDouble();

    float scale = i / size;
    sample *= lerp(0.1f, 1.0f, scale * scale);

    kernel.append(sample);
  }
}
```

In order for the kernel samples to fall within a hemisphere, the x and y components of each sample are randomly generated in the range -1.0 and +1.0, while the z component is randomly selected in the range 0.0 and 1.0. Furthermore, the length of each sample is scaled with a quadratic interpolation factor, in order to distribute a greater number of samples near the origin of the hemisphere, ie the current fragment.

Generating a sample kernel for each fragment is computationally too heavy. To overcome this problem, a single sample kernel is generated that is randomly rotated around the normal of each fragment. But similarly, generating a rotation vector for each fragment is too expensive. The solution to this problem is to create a small array of rotation vectors and then distribute this pattern of rotations along the entire screen-space.

Listing 3.4 : SSAO noise definition

```
void SetNoise(int w, int h)
{
  QRandomGenerator generator = QRandomGenerator::securelySeeded();

  noise.clear();

  for (int i = 0; i < NoiseSize * NoiseSize; i++)
  {
    float x = generator.generateDouble() * 2.0f - 1.0f;
    float y = generator.generateDouble() * 2.0f - 1.0f;
    noise.append({x, y, 0});
  }

  SetNoiseScale(w, h);
}

void SetNoiseScale(int w, int h)
{
  NoiseScale.setX(w);
  NoiseScale.setY(h);
  NoiseScale /= NoiseSize;
}
```

In order for the sample kernel rotations to be around the z axis (tangent space), the x and y components of each rotation vector are randomly generated in the range -1.0 and +1.0, while the z component is set to 0. The scale factor of the rotation pattern is also updated to adapt it to the current screen size. This scale factor will be used later in the fragment shader for the calculation of the sampling offsets of the rotation vectors.

The sample kernel and the rotation pattern are both transferred to the GPU through the use of textures, and together with the center and normal textures generated in the geometry step, they are the input of the occlusion step fragment shader.

Listing 3.5 : occlusion fragment shader

```
#version 450 core

struct FragData
{
  vec3 center;
  vec3 normal;
  vec3 noise;
};

smooth in vec2 offset;

uniform mat4 projection;

uniform sampler2D center;
uniform sampler2D normal;
uniform sampler2D kernel;
uniform sampler2D noise;

uniform int size; // kernel size
uniform vec2 scale; // noise scale
uniform float radius;
uniform float bias;

out float occlusion;

void main()
{
  FragData frag;
  frag.center = texture(center, offset).xyz;
  frag.normal = normalize(texture(normal, offset).xyz);
  frag.noise = normalize(texture(noise, offset * scale).xyz);

  // TBN matrix
  vec3 tangent = normalize(frag.noise - frag.normal * dot(frag.noise,
frag.normal));
  vec3 bitangent = cross(frag.normal, tangent);
  mat3 TBN = mat3(tangent, bitangent, frag.normal);

  occlusion = 0.0f;

  for(int i = 0; i < size; i++)
  {
    float x = float(i) / float(size);

    for(int j = 0; j < size; j++)
    {
      float y = float(j) / float(size);

      vec3 KernelSample = TBN * texture(kernel, vec2(x, y)).xyz;
      KernelSample = frag.center + KernelSample * radius;

      vec4 offset = vec4(KernelSample, 1.0f);
      offset = projection * offset;
      offset.xyz /= offset.w;
      offset.xyz = offset.xyz * 0.5f + 0.5f;
```

```
      float depth = texture(center, offset.xy).z;

      float range = smoothstep(0.0f, 1.0f, radius / abs(frag.center.z - depth));
      occlusion += ((depth >= KernelSample.z + bias) ? 1.0f : 0.0f) * range;
    }
  }

  occlusion = 1.0f - (occlusion / float(size * size));
}
```

The data of each fragment is extracted from the textures passed in input to the fragment shader.

With the normal and the rotation vector, a TBN (tangent, bitangent and normal) matrix is constructed by means of the Gram-Schmidt orthogonalization process to transform the samples (slightly inclined with respect to the rotation vector) from tanget-space to view-space.

Each kernel sample is then transformed into view-space, translated with respect to the fragment position and modulated by the uniform radius, which parameterizes the range of the SSAO.

The next step transforms the sample from view-space to screen-space (using the projection matrix, also passed to the fragment shader as uniform) and retrieve from the texture center (which contains the position in view-space of each fragment) the depth of the texel associated with the current sample.

This sample contributes to the occlusion factor if its depth is greater than or equal to the z component of the sample. In other words, the sample contributes to the occlusion factor, if it is "inside" some geometry near the fragment. Vice versa, if the depth is less than the z component of the sample, it means that in that particular position there are no elements of occlusion for the fragment.

```
occlusion += (depth >= KernelSample.z) ? 1.0f : 0.0f;
```

Without further precautions, this approach suffers from a problem when the fragment under examination is on the edge of a surface. In this case the depth values of surfaces that do not contribute to the occlusion factor are also taken into consideration, as the following image shows.

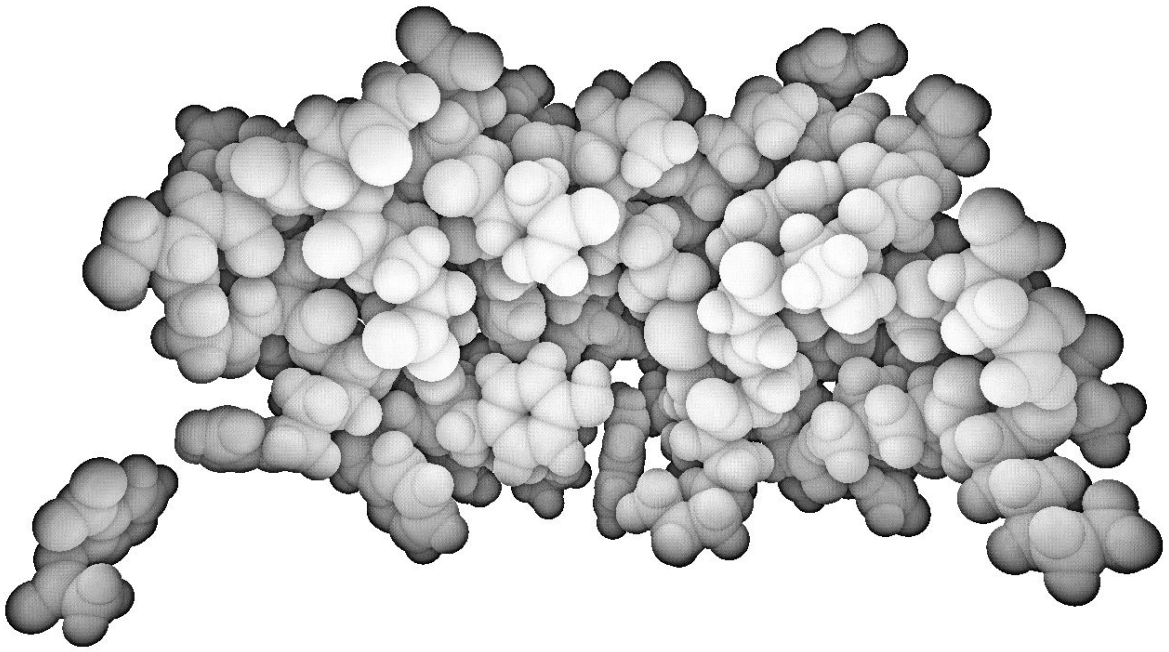


Figure 3.8 : bad occlusion factor

This problem can be prevented by checking if the sample depth is within the sampling radius. Moreover, instead of a brusque "on/off", the contribution of each sample is interpolated between 0 and 1 to ensure a smooth effect near the radius limit.

Finally, the fragment shader returns the normalized occlusion factor with respect to the sample kernel size.

Figure 3.9 : good occlusion factor

As can be seen from the figure above, the occlusion factor generated by the fragment shader allows us to perceive the depth of the model.

3.2.3 Blur Step

To get around the problem of having to generate a rotation vector for the sample kernel for each fragment, we created a small array of rotation vectors and used its values repeatedly along the whole screen-space.

This strategy has a price, observable in the Figure 3.9. The repeated pattern of the rotation vectors is distinctly recognizable. For this reason, before attenuating the ambient component in the lighting step with the occlusion factor, a blur effect should be applied to the latter.

Listing 3.6 : blur fragment shader

```
#version 450 core

smooth in vec2 coords;

uniform sampler2D occlusion;
uniform int NoiseSize;

out float blur;

void main()
{
  int HalfNoiseSize = int(NoiseSize * 0.5f);
  vec2 size = 1.0f / vec2(textureSize(occlusion, 0));

  blur = 0.0f;

  for (int x = -HalfNoiseSize; x < +HalfNoiseSize; x++)
  {
    for (int y = -HalfNoiseSize; y < +HalfNoiseSize; y++)
    {
      vec2 offset = vec2(x, y) * size;
      blur += texture(occlusion, coords + offset).r;
    }
  }

  blur *= 1.0f / (NoiseSize * NoiseSize);
}
```

The fragment shader receives the texture with the values of the occlusion factor generated by the previous step and uses the randomness pattern of the rotation vectors to generate a simple but effective blur effect.

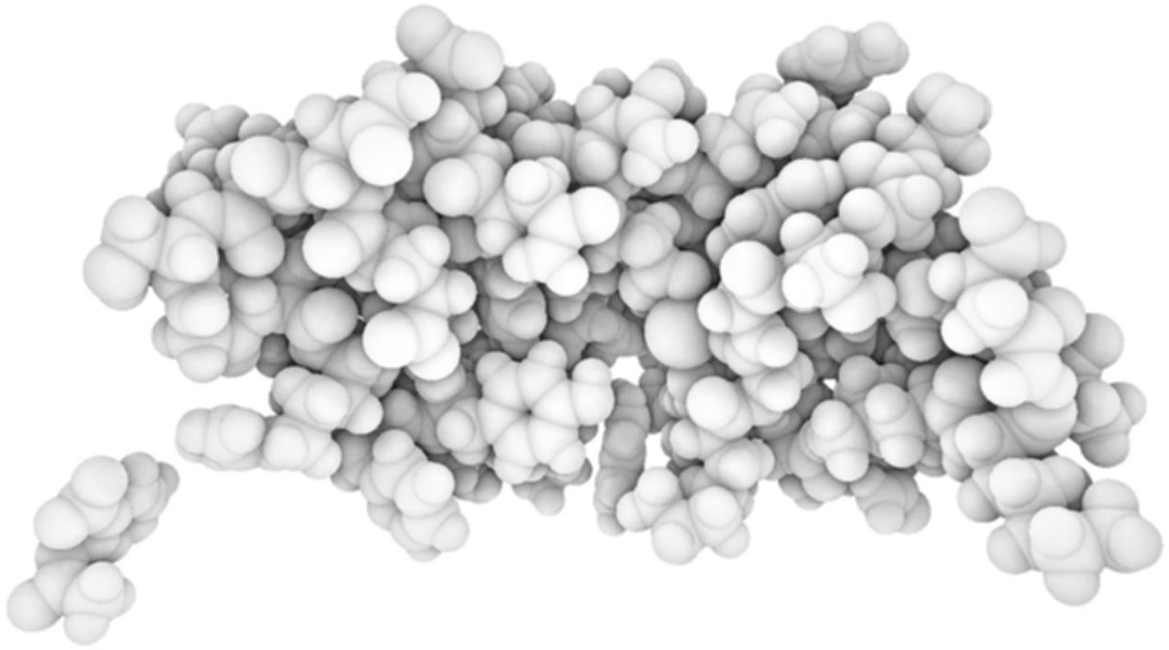


Figure 3.9 : occlusion factor with blur effect

The figure above shows the occlusion factor with the blur effect applied and that will be used in the next step.

3.2.4 Lighting Step

To integrate the occlusion factor into the lighting model, it is sufficient to attenuate the ambient component of the lighting model with the occlusion factor generated in the previous steps.

Listing 3.7 : lighting fragment shader

```
#version 450 core

struct Fragment
{
  vec3 center;
  vec3 normal;
  vec3 albedo;
  float SSAO;
};

struct Light
{
  // ...
};

void SetFragment(out Fragment frag);
vec3 GetLighting(Fragment frag, Light light, vec3 eye);

smooth in vec2 offset;
```

```
uniform sampler2D center;
uniform sampler2D normal;
uniform sampler2D albedo;
uniform sampler2D occlusion;

uniform float shininess;
uniform Light light;

out vec4 PixelColor;

void main()
{
  Fragment frag;
  SetFragment(frag);

  PixelColor = vec4(GetLighting(frag, light, vec3(0.0f)), 1.0f);
}

void SetFragment(out FragData frag)
{
  frag.center = texture(center, offset).xyz;
  frag.normal = texture(normal, offset).xyz;
  frag.albedo = texture(albedo, offset).rgb;
  frag.SSAO = texture(occlusion, offset).r;
}

vec3 GetLighting(Fragment frag, Light light, vec3 eye)
{
  float factor;

  // ambient component
  factor = frag.SSAO;
  vec3 ambient = factor * light.ambient;

  // diffuse component
  // ...

  // specular component
  // ...

  vec3 lighting = (ambient + diffuse + specular) * frag.albedo;
  return lighting;
}
```

The fragment shader of the lighting step receives the geometry and the colors of the scene as input through the textures generated in the geometry step, together with the values of the occlusion factor generated in the previous stages. Ambient occlusion is applied to each fragment simply by multiplying the ambient component of the lighting model by the occlusion factor extracted from the occlusion texture.

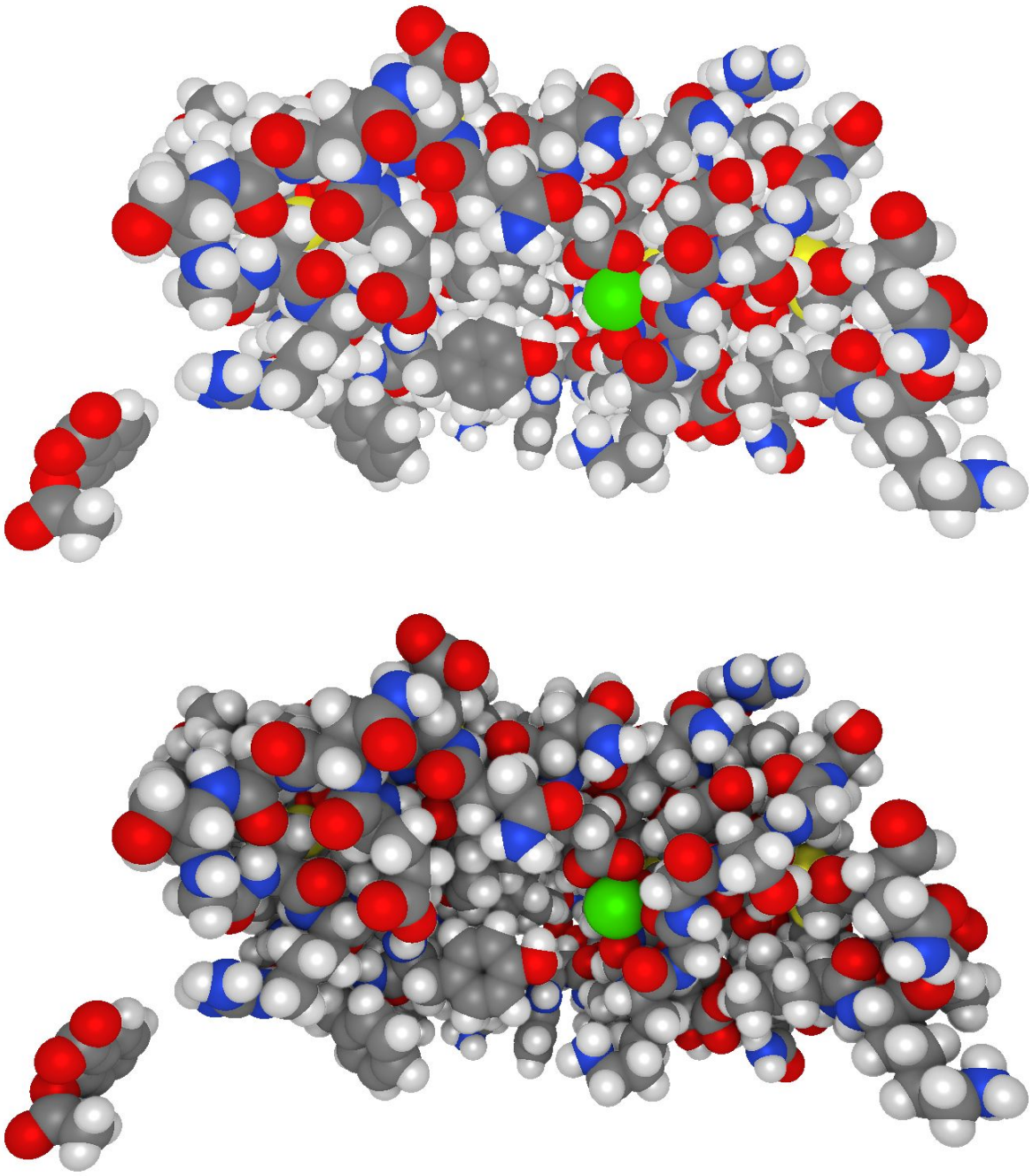


Figure 3.10 : the same scene without (top) and with (bottom) SSAO

In the figure above, the same scene is rendered without (on the left side) and with (on the right side) applying the occlusion factor.

As can be seen in Figure 3.10, SSAO raises the sense of depth returned by the scene, but it is also a computationally expensive technique. For this reason, in our application at any time the user can activate or deactivate the SSAO simply by toggling a checkbox.

4 Object Outlining

In Chapter 2 we have implemented an approach that allows us to visualize a large number of spheres (to represent the atoms of our molecules) with no tradeoff between performance and image quality.

In Chapter 3 we have increased the sense of realism transmitted by the rendered scene with the integration of a lighting model into the project.

This chapter shows how we implemented the feature that we will then use to visualize RMSD and RMSF of atoms and residues, that is the possibility to outline a collection of objects in the scene and to assign different outline colors to different objects.

4.1 Silhouette & Outline

The object outlining method used in the project consists of three phases or steps.

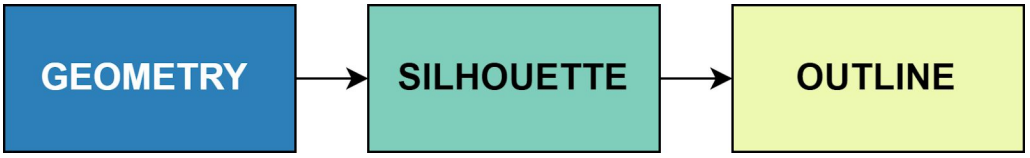


Figure 4.1 : steps of the object outlining method used in the project

- **Geometry Step** : this step is the same one used previously by SSAO to "export" the geometry of the scene, so it does not add any cost
- **Silhouette Step** : in this step the silhouette of the object of the scene to be outlined is drawn in a texture
- **Outline Step** : in this step the silhouette returned from the previous stage is used to define the outline, which is impressed on the object

4.1.1 Silhouette Step

The geometry step is the same one used previously by SSAO to "export" the geometry of the scene, so we can simply bind the appropriate framebuffer and copy the information we need.

Listing 4.1 : silhouette step

```
if (outline.active)
{
  glBindFramebuffer(GL_READ_FRAMEBUFFER, FBO::GEOMETRY);
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, FBO::SILHOUETTE);
  {
```

```
    int w = geometry().width();
    int h = geometry().height();
    GLbitfield mask = GL_DEPTH_BUFFER_BIT;

    // set clear color
    SetClearColor(Qt::GlobalColor::white);

    // copy depth buffer
    glBlitFramebuffer(0, 0, w, h, 0, 0, w, h, mask, GL_NEAREST);
    // enable depth test
    glEnable(GL_DEPTH_TEST);
    // change depth test function
    glDepthFunc(GL_LEQUAL);

    // clear color buffer
    glClear(GL_COLOR_BUFFER_BIT);

    // draw silhouette
    DrawSilhouette();
  }
  glBindFramebuffer(GL_FRAMEBUFFER, defaultFramebufferObject());
}
```

We specify the geometry FBO from the SSAO step as the read framebuffer and the silhouette FBO as the write framebuffer. We set the clear color with the white color. We copy the content of the geometry depth buffer to the silhouette depth buffer. We enable the depth test and change the depth test function to GL_LEQUAL. We the clear color buffer, but of course not the depth buffer, otherwise we would lose all the previously copied values. And finally, we draw the object (or collection of objects) that we want to outline.

The sequence of steps applied to the depth buffer ensure that only the visible parts of the object (or collection of objects) we want to outline will be silhouetted.

Listing 4.2 : silhouette fragment shader

```
#version 450

struct Impostor
{
  vec3 center;
  vec3 normal;
};

void SetImpostor(out Impostor impostor);

in Fragment
{
  flat vec3 center;
  flat float radius;
  smooth vec2 offset;
} frag;

uniform mat4 projection;
```

```
out float silhouette;

void main()
{
  Impostor impostor;
  SetImpostor(impostor);

  // frag depth
  vec4 clip = projection * vec4(impostor.center, 1.0f);
  float depth = clip.z / clip.w;

  float near = gl_DepthRange.near;
  float far = gl_DepthRange.far;
  float diff = gl_DepthRange.diff;
  gl_FragDepth = ((diff * depth) + near + far) * 0.5f;

  silhouette = 0.0f;
}
```

The fragment shader is mostly the same as Listing 2.3, the main difference is that here we simply return the black color for the fragments that pass the depth test. So we don't need to know the fragment albedo.

4.1.2 Outline Step

The outline step is the last step of the draw iteration.

Listing 4.3 : outline step

```
// set clear color
SetClearColor(BackgroundColor);

// disable depth test
glDisable(GL_DEPTH_TEST);

// clear color buffer
glClear(GL_COLOR_BUFFER_BIT);

// draw lighting
DrawLighting();

if (outline.active)
{
  DrawOutline();
}
```

We set the clear color with the desired background color, we disable the depth test and clear the color buffer. We draw the final scene returned by the lighting steps, and then over it (the depth test is still disabled) we draw the outline.

Listing 4.4 : outline fragment shader

```
#version 450

struct Outline
{
  int thickness;
  vec3 color;
};

in smooth vec2 texcoord;

uniform sampler2D silhouette;
uniform Outline outline;

out vec4 PixelColor;

void main()
{
  // if the texel is black (we are on the silhouette)
  if (texture(silhouette, texcoord).xyz == vec3(0.0f))
  {
    vec2 size = 1.0f / textureSize(silhouette, 0);

    for (int i = -outline.thickness; i <= +outline.thickness; i++)
    {
      for (int j = -outline.thickness; j <= +outline.thickness; j++)
      {
        if (i == 0 && j == 0)
        {
          continue;
        }

        vec2 offset = vec2(i, j) * size;

        // and if one of the texel-neighbor is white (we are on the border)
        if (texture(silhouette, texcoord + offset).xyz == vec3(1.0f))
        {
          PixelColor = vec4(outline.color, 1.0f);
          return;
        }
      }
    }
  }

  discard;
}
```

The outline fragment shader takes as inputs the texture silhouette produced by the previous step and the desired thickness and color for the outline. For each texel of the silhouette we check first if we are on the object to be outlined. If not, we can immediately discard the fragment. If instead we are on the object, we explore the neighbor texels in a radius equal to the desired outline thickness in search of a white texel (the clear color previously set in the silhouette step). If we find at least one white texel, it means that we have reached the edge of the object and we can then return the desired outline color. Otherwise, if the search fails, we discard the fragment.

Observe how this approach draws the outline always and only over the object to be outlined, without ever exceeding its edge. This effect is desired and will be exploited in the next section to outline different objects (atoms and residues of our molecules) with different colors.

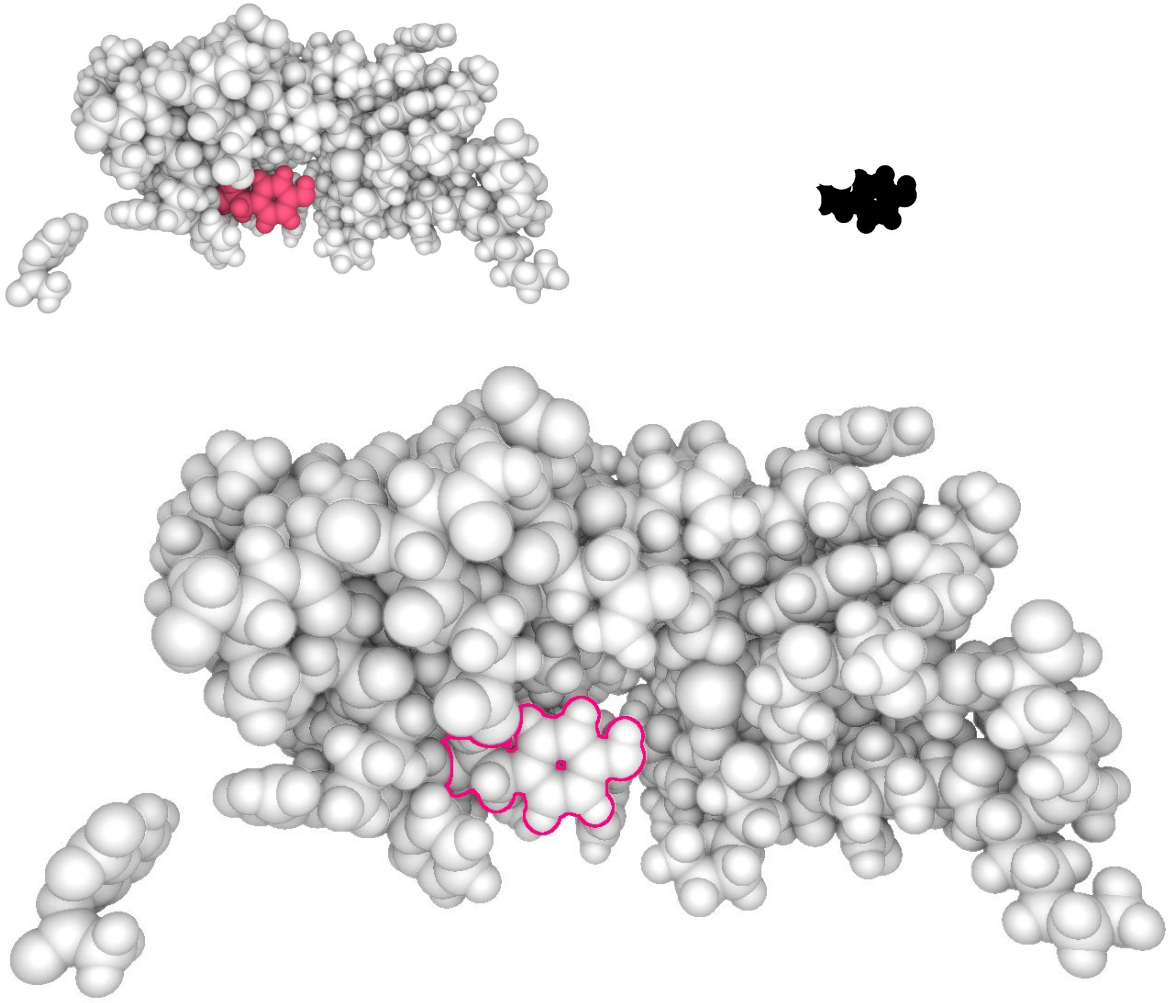


Figure 4.2 : steps of the outlining method

The figure above shows the steps of the outlining method. At the top-left corner, the residue we want to outline is highlighted in red, at the top-right corner, the silhouette of the residue and at the bottom, the outlined residue.

4.2 Multi-Outline for Atoms & Residues

The strategy illustrated in the previous section works well if we want to outline a collection of objects all with the same color, but it does not allow us to assign different colors to different objects.

4.2.1 Silhouette Step

To achieve the multi-outline effect, we first need to assign to the silhouette of each object (in our case atoms or residues) a unique color, so we can then associate the desired outline color to each object in the outline step.

A unique color for each object can be obtained by converting the serial number of atoms and residues into a triplet of bytes and use these values to define the red, green and blue channels of the object's color outline.

We can also take advantage of the fact that serial numbers of atoms and residuals are always positive values. This means that the byte triplet (0, 0, 0) is not associated with any object and therefore the black color can be used as clear color for the silhouette.

Listing 4.5 : multi-silhouette vertex shader

```
#version 450 core

layout (location = 0) in vec3 center;
layout (location = 1) in float radius;
layout (location = 2) in vec3 albedo;
layout (location = 3) in vec2 number;

uniform int OutlineMode;

out VertData
{
  vec3 center;
  float radius;
  vec3 colour;
} vert;

void main()
{
  uvec3 mask = uvec3(0x00FF0000, 0x0000FF00, 0x000000FF);
  uvec3 shift = uvec3(16, 8, 0);

  vec3 colour;

  switch (OutlineMode)
  {
  // residue number
  case 0:
  case 1:
  {
    colour = ((uvec3(number[1]) & mask) >> shift) / 255.0f;
    break;
  }
  // atom number
  case 2:
  case 3:
  {
```

```
    colour = ((uvec3(number[0]) & mask) >> shift) / 255.0f;
    break;
  }
  }

  vert.center = center;
  vert.radius = radius;
  vert.colour = colour;
}
```

The new silhouette vertex shader receives two additional information, namely the pair of atom and residue serial numbers and the OutlineMode uniform to determine which of the two serials should be used to define the color of the object's silhouette.

The conversion between serial number and byte triplet is computed with a sequence of bitwise operations. The result is then divided by 255 to obtain a triplet of values in the range 0 - 1, that is to say the color to be assigned to the object's silhouette in the silhouette fragment shader.

Figure 4.3 : silhouette colored based on the residue numbers

The figure above shows the colored silhouette based on the residue numbers. It should not be surprising to observe dark blue gradations. The number of residues in our case study is 121 and the first 255 values are mapped to the blue channel.

4.2.2 Outline Step

Before drawing the outline, we pass to the multi-outline fragment shader a 1D texture with the outline colors of all atoms or residues, depending on the outline mode. The colors in the

texture are ordered based on the serial numbers of atoms or residues. We can therefore access the outline color of an object by indexing the texture directly with its serial.

Listing 4.6 : multi-outline fragment shader

```
#version 450

struct Outline
{
  int thickness;
  sampler1D color;
};

in smooth vec2 texcoord;

uniform sampler2D silhouette;
uniform Outline outline;

out vec4 PixelColor;

void main()
{
  vec3 texel = texture(silhouette, texcoord).rgb;

  // if the texel is NOT black (we are on the silhouette)
  if (texel != vec3(0.0f))
  {
    vec2 size = 1.0f / textureSize(silhouette, 0);

    for (int i = -outline.thickness; i <= +outline.thickness; i++)
    {
      for (int j = -outline.thickness; j <= +outline.thickness; j++)
      {
        if (i == 0 && j == 0)
        {
          continue;
        }

        vec2 offset = vec2(i, j) * size;

        // and if one of the texel-neighbors has a different color
        // (we are on the border or on another atom/residue)
        if (texture(silhouette, texcoord + offset).rgb != texel)
        {
          // retrieve atom/residue number
          uvec3 shift = uvec3(16, 8, 0);
          uvec3 rgb = uvec3(texel * 255.0f);

          uint number = 0;
          for (int j = 0; j < 3; j++)
          {
            number += rgb[j] << shift[j];
          }

          // retrieve outline color
          float size = 1.0f / textureSize(outline.color, 0);
```

```
          float offset = number * size + size * 0.5f;
          vec3 color = texture(outline.color, offset).rgb;

          PixelColor = vec4(color, 1.0f);
          return;
        }
      }
    }
  }

  discard;
}
```

The multi-outline fragment shader is mostly the same as Listing 4.4. We first sample the silhouette texture and check if the texel color is black. If it is, we are not on the silhouette and we can directly discard the fragment. Otherwise, as before, we explore the neighboring texels in a radius equal to the desired outline thickness in search of a texel whose color is different from that of the current texel. If the search fails, we discard the fragment. Otherwise, if there is any, it means that we have reached the edge of the object or the silhouette of another object. In both cases, we retrieve the serial number reconverting the texel color and use it to offset the outline texture color. We can then return the desired outline color.

4.3 Outline Color Mapping

Choose the best coloring for the outline is a matter of tweaking and experimentation.

4.3.1 Color Palettes

The application gives the user the possibility to choose between different **color brewer** palettes.

Listing 4.7 : outline color palettes

```
enum ColorPaletteType { sequential, diverging, qualitative };

struct ColorPalette
{
  QString name;
  ColorPaletteType type;
  QMap<int, QString> colors;
};

QVector<ColorPalette> ColorPalettes
{
  {
    "RdPu",
    sequential,
    {
      {3, "#fde0dd #fa9fb5 #c51b8a"},
      {4, "#feebe2 #fbb4b9 #f768a1 #ae017e"},
      {5, "#feebe2 #fbb4b9 #f768a1 #c51b8a #7a0177"},
```

```
      {6, "#feebe2 #fcc5c0 #fa9fb5 #f768a1 #c51b8a #7a0177"},
      {7, "#feebe2 #fcc5c0 #fa9fb5 #f768a1 #dd3497 #ae017e #7a0177"}
    }
  },
  {
    "BrBG",
    diverging,
    {
      {3, "#d8b365 #f5f5f5 #5ab4ac"},
      {4, "#a6611a #dfc27d #80cdc1 #018571"},
      {5, "#a6611a #dfc27d #f5f5f5 #80cdc1 #018571"},
      {6, "#8c510a #d8b365 #f6e8c3 #c7eae5 #5ab4ac #01665e"},
      {7, "#8c510a #d8b365 #f6e8c3 #f5f5f5 #c7eae5 #5ab4ac #01665e"}
    }
  },
  {
    "RdYlBu",
    diverging,
    {
      {3, "#fc8d59 #ffffbf #91bfdb"},
      {4, "#d7191c #fdae61 #abd9e9 #2c7bb6"},
      {5, "#d7191c #fdae61 #ffffbf #abd9e9 #2c7bb6"},
      {6, "#d73027 #fc8d59 #fee090 #e0f3f8 #91bfdb #4575b4"},
      {7, "#d73027 #fc8d59 #fee090 #ffffbf #e0f3f8 #91bfdb #4575b4"}
    }
  },
};
```

By filling the ColorPalettes array with additional ColorPalette structures, the new palettes will be automatically integrated into the application and made available to the user.

4.3.2 Color Scheme

The outline colors are organized in color schemes, which are a collection of color steps. Each color spet defines a values range (minimum and maximum values) and a color associated with that range.

When a value falls within the range defined by the color step, the color of the step is assigned to the outline.

Listing 4.8 : outline color scheme

```
struct ColorStep
{
  int number;
  float min;
  float max;
  QColor color;
};

typedef QVector<ColorStep> ColorScheme;

ColorScheme GetColorScheme(float min, float max, ColorPalette palette, int size)
```

```
{
  ColorScheme scheme;

  QStringList colors = palette.colors[size].split(" ");
  float span = (max - min) / size;

  for (int i = 0; i < size; i++)
  {
    ColorStep step;
    step.number = i;
    step.min = min + span * i;
    step.max = min + span * (i+1);
    step.color = colors[i];
    scheme += step;
  }

  scheme.first().min = 0;
  scheme.last().max = FLT_MAX;

  return scheme;
}

static ColorStep GetColorStep(ColorScheme scheme, float value)
{
  for (auto step : scheme)
  {
    if (step.min <= value && value < step.max)
    {
      return step;
    }
  }

  ColorStep InvalidStep;
  InvalidStep.number = -1;
  InvalidStep.min = NAN;
  InvalidStep.max = NAN;
  InvalidStep.color = QColor::Invalid;

  return InvalidStep;
}
```

The GetColorScheme() method takes the minimum and maximum values of the range to be partitioned and the number of desired partitions (size). It constructs a color scheme by partitioning the range of values in the specified number of partitions and associating the corresponding colors extracted from the selected palette.

The GetColorStep() method receives a color scheme and a value to be tested. It checks if the value falls into one of the color scheme partitions and if so, returns the corresponding color step, from which the outline color can be extracted.

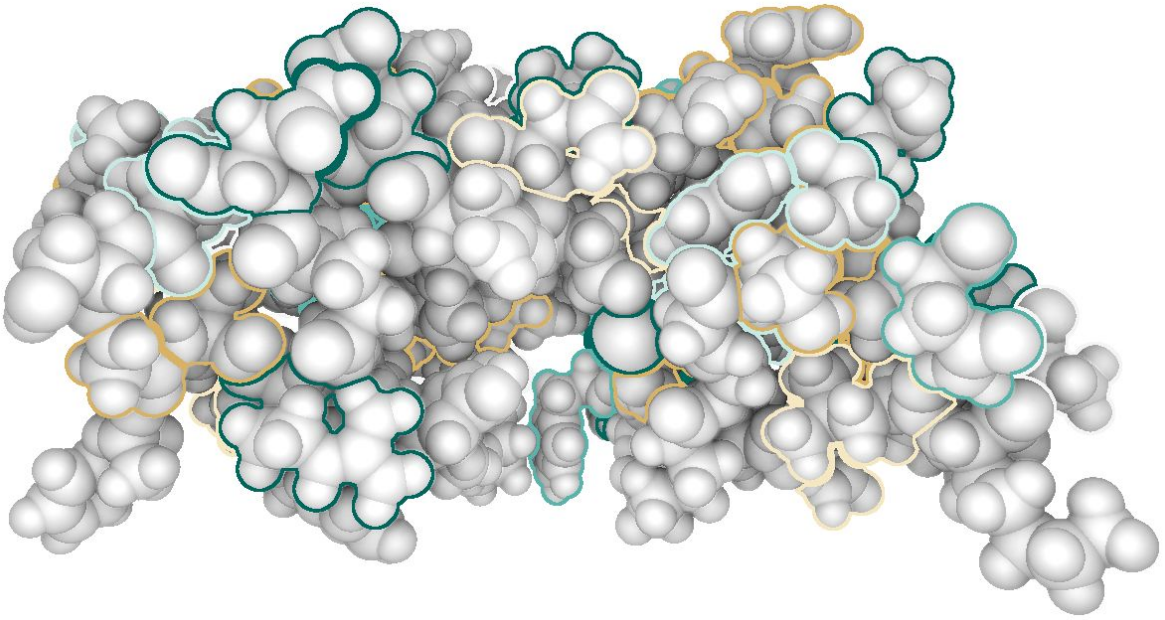


Figure 4.4 : multi-outlining with color brewer palette

The figure above shows the multi-outlining effect achieved by mapping a color brewer palette.

In the next chapter we will calculate the values of RMSD and RMSF with which we will define the outline color schemes for atoms and residues.

5 RMSD & RMSF

In Chapters 2 and 3 we built a 3D viewer to display molecular models. We have chosen space-filling as method of representation. This means that each atom of the molecule is represented by a sphere and, using a technique called impostors, we did not have to compromise between performance and image quality. Moreover, in order to increase the sense of realism of the model, we have integrated a lighting model into the application.

In the previous chapter we implemented the multi-outlining for atoms and residues. This feature allows us to outline each atom or residue of the molecule and assign a different outline color to each of them.

This chapter illustrates how the values of root mean square deviation (RMSD) and root mean square fluctuation (RMSF) have been computed, both at atom and residue level, and how they are mapped to the outline color of the objects in the scene.

5.1 Deviations & Fluctuations

The root mean square deviation (or RMSD) of atomic positions is the measure of the average distance between the atoms of superimposed molecules, or, more generally, it is a measure of the average distance between two different conformations of a set of points in three-dimensional space. In other words, the RMSD expresses how different two conformations are from each other.

We can express the RMSD with the following equation.

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \left\| x_{i,\,t} - x_{i,\,t_{ref}} \right\|^2}$$

Equation 5.1 : root mean square deviation

In Equation 5.1, N is the number of points in the set and $x_{i,t}$ represents the position of the point i at the time t. The set of positions of all the points in the set at a given time t is called **conformation** at the time t. The conformation at the time t_{ref} is called **reference conformation** and is used as a reference value to measure the deviation of another conformation.

When a dynamical system (like a molecule) fluctuates about some well-defined reference position, the RMSD from the average over time can be referred to as the root mean square fluctuation (or RMSF).

The RMSF is then a measure of the deviation of the position of a particle with respect to a reference position over time. Therefore this quantity can measure the average fluctuation of an atom around a reference position.

We can express the RMSF with the following equation.

$$RMSF = \sqrt{\frac{1}{T}\sum_{t=1}^{T} \left\| x_{i,\,t} - x_{i,\,t_{ref}} \right\|^2}$$

Equation 5.2 : root mean square fluctuation

In Equation 5.2, T is the samples number of the positions of the points in the set (or the time instants) along which the fluctuation is calculated. As before, $x_{i,\,t}$ represents the position of the point i at the time t and $x_{i,\,t_{ref}}$ represents its reference position.

When used to measure the deviation or fluctuation of atomic positions, RMSD and RMSF are typically expressed in **angstroms**, where 1 Å is equal to 10^{-10} m.

5.2 Comparing Conformations

When we compare the conformations of a dynamic system, such as the molecular conformations sampled from a molecular simulation, before applying Equation 5.1 and Equation 5.2 on a conformation to calculate its RMSD and RMSF, a **roto-translation** on the conformation should be carried out to align it as best as possible to the reference conformation.

We apply this transformation on the conformation in order to compensate the fact that a molecule can (and probably will) drift away from its origin and rotate in an arbitrary way during its trajectory.

The result will be the minimum (or least) RMSD and RMSF over all the possible relative positions and orientations of the two conformations under consideration.

In this way we obtain a measure of the deviations and fluctuations mostly due to the interactions between the molecules and that tries to filter out other possible turbulences due to the independent movement of the structures.

5.2.1 Proper Rigid Transformation

Before calculating the RMSD and RMSF of a conformation, we will apply a **roto-translation** to align it as best as possible to the reference configuration.

Each roto-translation (or **proper rigid transformation**) can be decomposed into two operations: a translation and a rotation. This class of transformations has an important property: any object will keep the same shape and size after a proper **rigid transformation**.

It is a subclass of the class transformations that also includes the operation of reflection and whose elements are called rigid transformations. Any rigid transformation preserves the of the objects size (the euclidean distance between every pair of points), but in general does not preserve the objects shape (due to the possible reflections).

A rigid transformation is formally defined as a transformation that, when acting on any vector $\vec{x}$, produces a transformed vector $\vec{x}'$ of the following form.

$$\vec{x}' = R\,\vec{x} + \vec{t}$$

Equation 5.3 : rigid transformation

Where R is an orthogonal transformation (i.e., $R^T = R^{-1}$), and $\vec{t}$ is a vector giving the translation of the origin.

A proper rigid transformation also satisfies the following property.

$$det(R) = 1$$

Equation 5.4 : proper rigid transformation

Which means that R does not produce a reflection, and hence it represents a rotation (an orientation-preserving orthogonal transformation).

Indeed, when an orthogonal transformation matrix produces a reflection, its determinant is equal to –1.

5.2.2 Optimal Alignment & Kabsch Algorithm

As stated in the previous paragraph, the optimal alignment requires both a translation and a rotation.

We can resolve the translation part if we first move the conformation so that its centroid coincides with the centroid of the reference conformation.

The **centroid** of a set of points is the average position of all its points.

$$centroid = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Equation 5.5 : centroid equation

We still have to determine the optimal rotation that minimize RMSD and RMSF. Fortunately, it is a well-studied problem and several solutions have been proposed.

One solution is known as the **kabsch algorithm**, and it works as follows.

Let $\{a_i\}$ and $\{b_i\}$ be two sets of points defined in the three-dimensional space both of cardinality N.

- First we translate both sets of points, so that their centroid coincides with the origin of the coordinate system. This is done by subtracting from the point coordinates the coordinates of the respective centroid.

- We represent both set of points with the $N \times 3$ matrices A and B, where for both matrices the i-th row represents the position of the i-th point in the respective set.

- We calculate the **covariance matrix** as $C = A^T B$.

- We calculate the **singular value decomposition** (or SVD) of the covariance matrix $C = USV^T$.

- We check if we need to correct our rotation matrix to ensure a proper rigid transformation testing the determinant's sign of the matrix VU^T.

$$d = det(VU^T)$$

Equation 5.6 : check for proper rigid transformation

- Finally, we calculate our optimal rotation matrix R, as

$$R = V \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & d \end{pmatrix} U^T$$

Equation 5.7 : optimal rotation matrix

The following figure shows a conformation before and after the alignment operation.

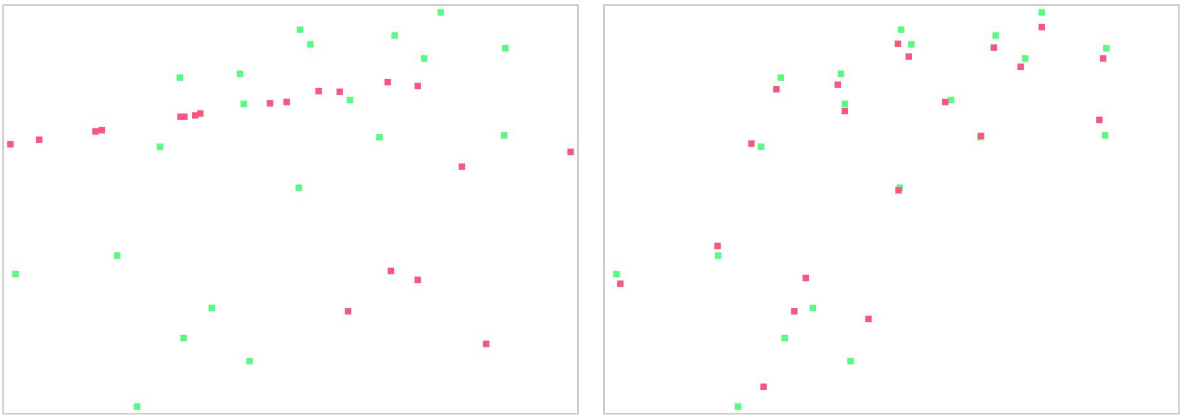


Figure 5.1 : same conformation not aligned (left) and optimally aligned (right)

The points in green represent the reference conformation, while the points in red represent the conformation that we want to align. Both conformations are translated so that their centroid coincides with the origin of the coordinate system. The left side of the figure shows the non-aligned conformation, while the right side shows the conformation aligned using the optimal rotation matrix.

5.2.3 Minimum RMSD & RMSF

We can now update the equations of RMSD and RMSF (Equation 5.1 and Equation 5.2) to obtain their respective minimum values.

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \left\| R\left(x_{i,t} - c_t\right) - \left(x_{i,t_{ref}} - c_{t_{ref}}\right) \right\|^2}$$

Equation 5.8 : least root mean square deviation

$$RMSF = \sqrt{\frac{1}{T}\sum_{t=1}^{T} \left\| R\left(x_{i,t} - c_t\right) - \left(x_{i,t_{ref}} - c_{t_{ref}}\right) \right\|^2}$$

Equation 5.9 : least root mean square fluctuation

The new equations differ from the previous ones in two aspects.

- The two conformations are translated so that their centroid coincides with the origin of the coordinate system, by subtracting from the point coordinates the coordinates of the respective centroid.

- The conformation at time t is aligned as best as possible to the reference configuration, by multiplying its point coordinates by the optimal rotation matrix R.

5.3 Code Implementation

In this last section of the chapter, we will see how the equations and the kabsch algorithm presented in the previous section have been translated into code to be integrated into the project.

5.3.1 Class Definitions

First, we define the Atom, Residue and Trajectory classes.

Listing 5.1 : atom class

```
class Atom
{
  int number;
  QString name;
  QString element;
  int residue;

  QVector<float> RMSDs;
  float MinRMSD;
  float MaxRMSD;

  float RMSF;
};
```

Listing 5.2 : residue class

```
class Residue
{
  int number;
  QString chain;
  QString name;
  int sequence;
  QVector<int> atoms;

  QVector<float> RMSDs;
  float MinRMSD;
  float MaxRMSD;

  float RMSF;
};
```

The atom and residue classes have, among others, these attributes.

- **RMSDs** : an array to store all the RMSD values, one value for each step of molecular simulation
- **MinRMSD** / **MaxRMSD** : minimum and maximum RMSD values per atom / residue
- **RMSF** : a unique RMSF value calculated for the entire molecular simulation

Listing 5.3 : trajectory class

```
using namespace Eigen; // Vector3f, Matrix3f

class Trajectory
{
    // map <atom number, atom position>
    typedef QMap<int, QVector3D> Model;

    // cooked data
    QMap<int, Atom> atoms;
    QMap<int, Residue> residues;
    QVector<Model> models;

    // Min and Max init values
    const float MinInitValue = FLT_MAX;
    const float MaxInitValue = 0;

    // Min and Max residues RMSD
    float MinResiduesRMSD;
    float MaxResiduesRMSD;
    // Min and Max residues RMSF
    float MinResiduesRMSF;
    float MaxResiduesRMSF;

    // Min and Max atoms RMSD
    float MinAtomsRMSD;
    float MaxAtomsRMSD;
    // Min and Max atoms RMSF
    float MinAtomsRMSF;
    float MaxAtomsRMSF;

    QMap<int, QVector<QVector<Vector3f>>> ConformationLookupTable;
    QMap<int, QVector<Matrix3f>> OptimalRotationMatrixLookupTable;


    Matrix3f GetR(QVector<Vector3f> a, QVector<Vector3f> b);
    float GetRMSD(QVector<Vector3f> a, QVector<Vector3f> b, Matrix3f R);

    void CookResidues();
    void CookAtoms();

    void SaveCookedData();
    void LoadCookedData();
};
```

The trajectory class have, among others, these attributes and methods.

- **atoms** : a map <atom number, atom> with all the atoms of the simulation
- **residues** : a map <residue number, residue> with all the residues of the simulations

- **models** : an array of maps <atom number, atom position> with all the steps / configurations of the simulation
- **min / max values** : minimum and maximum RMSD and RMSF values of atoms and residues on the whole simulation
- **lookup tables** : supporting data structures to reduce the number of computations
- **GetR()** : method that returns the optimal rotation matrix between two conformations
- **GetRMSD()** : method that returns the RMSD between two conformations
- **CookResidues()** : method that calculates RMSD and RMSF for all the residues in the simulation
- **CookAtoms()** : method that calculates RMSD and RMSF for all the atoms in the simulation
- **save/load data methods** : methods to save and load the computed data

5.3.2 Method Definitions

We will now discuss the methods of the trajectory class.

The Qt framework supports matrices up to order four and it does not calculate the SVD of a matrix. To work with matrices of arbitrary size and for the SVD computation we will use the open source library **Eigen**.

Listing 5.4 : GetR() trajectory method

```
// a : reference conformation
// b : movement conformation
// both conformations should be centered with respect to their centroid
// the returned matrix is stored in column-major order
Matrix3f Trajectory::GetR(QVector<Vector3f> a, QVector<Vector3f> b)
{
  int size = a.size();
  assert(size == b.size());

  MatrixXf A(3, size);
  MatrixXf B(3, size);

  for (int i = 0; i < size; i++)
  {
    A.block(i, 0, 1, 3) << a[i].transpose();
    B.block(i, 0, 1, 3) << b[i].transpose();
  }

  // covariance matrix
  MatrixXf C = A.transpose() * B;

  // singular value decomposition
  JacobiSVD<MatrixXf> SVD(C, ComputeThinU | ComputeThinV);
```

```
  // left and right singular vectors
  Matrix3f U = SVD.matrixU();
  Matrix3f V = SVD.matrixV();

  // proper rotation test
  int d = signbit((V * U.transpose()).determinant()) ? -1 : +1;

  // optimal rotation
  Matrix3f D = DiagonalMatrix<float, 3>(1, 1, d);
  Matrix3f R = V * D * U.transpose();

  return R;
}
```

The GetR() method receives two conformations, applies the kabsch algorithm and returns the optimal rotation matrix. GetR() assumes that the centroid of both conformations coincides with the origin of the reference system.

Listing 5.5 : GetRMSD() trajectory method

```
// a : reference conformation
// b : movement conformation
// R : optimal rotation matrix
// both conformations should be centered with respect to their centroid
float Trajectory::GetRMSD(QVector<Vector3f> a, QVector<Vector3f> b, Matrix3f R)
{
  int size = a.size();
  assert(size == b.size());

  float RMSD = 0;
  for (int i = 0; i < size; i++)
  {
    RMSD += (R * b[i] - a[i]).squaredNorm();
  }
  RMSD /= size;

  return sqrt(RMSD);
}
```

The GetRMSD() method takes two conformations and the optimal rotation matrix between them, applies Equation 5.8 and returns the RMDS. GetRMSD() assumes that the centroid of both conformations coincides with the origin of the reference system.

Listing 5.6 : CookResidues() trajectory method

```
void Trajectory::CookResidues()
{
  auto FirstModel = models.first();

  // initialize residues Min and Max RMSD
  MinResiduesRMSD = MinInitValue;
```

```
  MaxResiduesRMSD = MaxInitValue;

  // initialize residues Min and Max RMSF
  MinResiduesRMSF = MinInitValue;
  MaxResiduesRMSF = MaxInitValue;

  for (auto &residue : residues)
  {
    // reference conformation : 1st model
    QVector<QVector3D> x;
    QVector<Vector3f> a;

    // get conformation atoms positions
    for (auto AtomNumber : residue.atoms)
    {
      x += FirstModel[AtomNumber];
    }

    // get conformation centroid
    QVector3D c = GetCentroid(x);

    for (auto &v : x)
    {
      // move centroid to reference system origin
      v = v - c;
      // from QVector3D to Vector3f
      a += FromQVector3DToVector3f(v);
    }
    ConformationLookupTable[residue.number] += a;

    // get conformation rotate matrix
    Matrix3f R = GetRotateMatrix(a, a);
    OptimalRotationMatrixLookupTable[residue.number] += R;

    // get conformation RMSD
    residue.RMSDs += GetRMSD(a, a, R);

    // initialize residue Min and Max RMSD
    residue.MinRMSD = MinInitValue;
    residue.MaxRMSD = MaxInitValue;

    for (auto model : models.mid(1))
    {
      // movement conformation
      QVector<QVector3D> y;
      QVector<Vector3f> b;

      // get conformation atoms positions
      for (auto AtomNumber : residue.atoms)
      {
        y += model[AtomNumber];
      }

      // get conformation centroid
      QVector3D c = GetCentroid(y);

      for (auto &v : y)
```

```
      {
        // move centroid to reference system origin
        v = v - c;
        // from QVector3D to Vector3f
        b += FromQVector3DToVector3f(v);
      }
      ConformationLookupTable[residue.number] += b;

      // get conformation rotate matrix
      Matrix3f R = GetRotateMatrix(a, b);
      OptimalRotationMatrixLookupTable[residue.number] += R;

      // get conformation RMSD
      float RMSD = GetRMSD(a, b, R);
      residue.RMSDs += RMSD;

      // update residue Min and Max RMSD
      residue.MinRMSD = (RMSD < residue.MinRMSD) ? RMSD : residue.MinRMSD;
      residue.MaxRMSD = (RMSD > residue.MaxRMSD) ? RMSD : residue.MaxRMSD;
    }

    // get residue RMSF
    residue.RMSF = GetAverage(residue.RMSDs);

    // update residues Min and Max RMSD
    MinResiduesRMSD = (residue.MinRMSD < MinResiduesRMSD) ? residue.MinRMSD :
MinResiduesRMSD;
    MaxResiduesRMSD = (residue.MaxRMSD > MaxResiduesRMSD) ? residue.MaxRMSD :
MaxResiduesRMSD;

    // update residues Min and Max RMSF
    MinResiduesRMSF = (residue.RMSF < MinResiduesRMSF) ? residue.RMSF :
MinResiduesRMSF;
    MaxResiduesRMSF = (residue.RMSF > MaxResiduesRMSF) ? residue.RMSF :
MaxResiduesRMSF;
  }
}
```

The method CookResidues() calculates the RMSD of each residue at each step of the simulation. The first step of the simulation is chosen as the reference conformation. For each residue the minimum and maximum RMSD values are saved. The RMSF of each residuel is calculated as an average on all RMSD values. Finally, the minimum and maximum RMSD values of all residues on the entire simulation are saved.

Listing 5.7 : CookAtoms() trajectory method

```
void Trajectory::CookAtoms()
{
  // initialize atoms Min and Max RMSD
  MinAtomsRMSD = MinInitValue;
  MaxAtomsRMSD = MaxInitValue;

  // initialize atoms Min and Max RMSF
  MinAtomsRMSF = MinInitValue;
```

```
  MaxAtomsRMSF = MaxInitValue;

  for (auto &atom : atoms)
  {
    // atom conformation index
    int AtomIndex = residues[atom.residue].atoms.indexOf(atom.number);

    // reference conformation : 1st model
    auto x = ConformationLookupTable[atom.residue].first();

    // get atom position
    Vector3f a = x[AtomIndex];

    // get rotate matrix
    Matrix3f R = OptimalRotationMatrixLookupTable[atom.residue][0];

    // get atom RMSD
    atom.RMSDs += (R * a - a).norm();

    // initialize atom Min and Max RMSD
    atom.MinRMSD = MinInitValue;
    atom.MaxRMSD = MaxInitValue;

    for (int i = 1; i < models.size(); i++)
    {
      // movement conformation
      auto y = ConformationLookupTable[atom.residue][i];

      // get atom position
      Vector3f b = y[AtomIndex];

      // get rotate matrix
      Matrix3f R = OptimalRotationMatrixLookupTable[atom.residue][i];

      // get atom RMSD
      float RMSD = (R * b - a).norm();
      atom.RMSDs += RMSD;

      // update atom Min and Max RMSD
      atom.MinRMSD = (RMSD < atom.MinRMSD) ? RMSD : atom.MinRMSD;
      atom.MaxRMSD = (RMSD > atom.MaxRMSD) ? RMSD : atom.MaxRMSD;
    }

    // get atom RMSF
    atom.RMSF = GetAverage(atom.RMSDs);

    // update atoms Min and Max RMSD
    MinAtomsRMSD = (atom.MinRMSD < MinAtomsRMSD) ? atom.MinRMSD : MinAtomsRMSD;
    MaxAtomsRMSD = (atom.MaxRMSD > MaxAtomsRMSD) ? atom.MaxRMSD : MaxAtomsRMSD;

    // update atoms Min and Max RMSF
    MinAtomsRMSF = (atom.RMSF < MinAtomsRMSF) ? atom.RMSF : MinAtomsRMSF;
    MaxAtomsRMSF = (atom.RMSF > MaxAtomsRMSF) ? atom.RMSF : MaxAtomsRMSF;
  }
}
```

The method CookAtoms() calculates the RMSD of each atom at each step of the simulation. The first step of the simulation is chosen as the reference conformation. For each atom the minimum and maximum RMSD values are saved. The RMSF of each atom is calculated as an average on all RMSD values. Finally, the minimum and maximum RMSD values of all atoms on the entire simulation are saved.

5.3.3 Data Storage

Computing all RMSD and RMSF values for each atom and residual at each step of the trajectory takes time, even for a relatively small simulation such as our case study.

It could be argued that the sequential strategy adopted here is not the most efficient way. Certainly more performing results could be obtained by parallelizing the RMSD calculation, given the intrinsic independence of the computation.

However, it was preferred to perform the calculation “offline” only once and to store the results in an external file, from which data is loaded when the application is started.

We can now use the calculated RMSD and RMSF values to define the outline color of atoms and residues.

6 User Interface

In this chapter we will describe the components of the graphical user interface (or UI) of our application and the features that it offers to the user.

The UI is implemented with the **Qt framework** and consists of three main components.

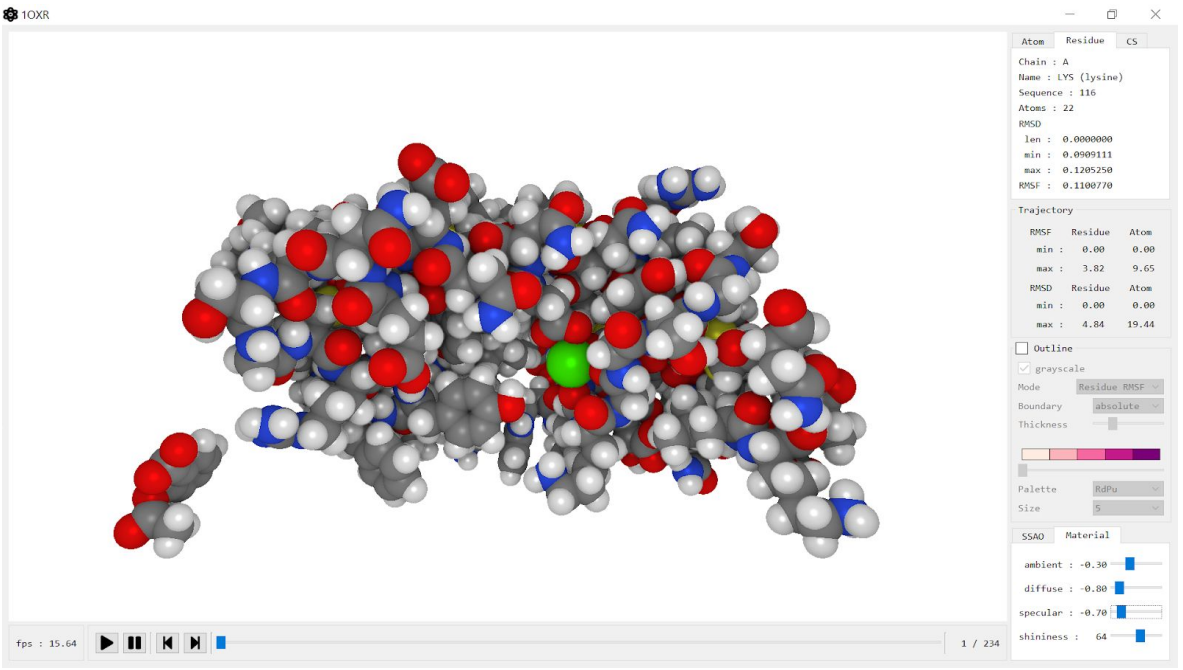


Figure 6.1 : user interface components

- **central area** : this is the OpenGL widget in which the molecular 3D models are rendered
- **right panel** : this widget is made up of different parts, some of which return information on the molecular simulation and others allow to configure the scene
- **bottom bar** : with this widget the user can move between the steps of the molecular simulation

In the following sections we will analyze each component of the UI in detail.

6.1 Central Area

As you can see in Figure 6.1, the area dedicated to rendering the scene takes up most of the application window.

This part of the UI consists of an OpenGL widget and, in addition to rendering the 3D molecular model of the current step in the simulation, allows the user to interact with the model through the use of the mouse.

Using the mouse on the OpenGL widget, the user can perform the following actions.

- **model rotating** : moving the mouse when the left button is pressed, the user can rotate the molecular model in object-space, that is around its axes

- **model panning** : moving the mouse when the middle button is pressed, the user can move the molecular model around the scene in view-space

- **camera orbiting** : moving the mouse when the right button is pressed, the user can rotate the camera around the origin of the world-space

- **camera zooming** : scrolling the mouse wheel, the user can move the camera closer or further away from the origin of the world-space

- **atom selection** : hovering the mouse cursor over an atom of the molecular model, the user can retrieve its data

We will take a look at how each of these actions is implemented in the application.

The status of each mouse button is stored with a flag. When the user presses a mouse button, the respective flag is activated and the coordinates of the current mouse cursor are stored. When the user releases a previously pressed mouse button, the respective flag is deactivated.

6.1.1 Model Rotating

The rotation of the molecular model, which is activated by moving the mouse when the left button is pressed, takes place in object-space, which means that the model is rotated around its own axes by updating the model matrix.

Listing 6.1 : mouse move handler - model rotating

```
void OpenGLWidget::mouseMoveEvent(QMouseEvent *event)
{
  int x = event->pos().x();
  int y = event->pos().y();
  float factor = 0.005f;

  // model rotating
  if (mouse.MouseLeftButton)
  {
    float dx = factor * (x - mouse.lastX);
    float dy = factor * (y - mouse.lastY);

    QMatrix4x4 rotation;
```

```
    rotation.rotate(qRadiansToDegrees(dx), camera.up);
    rotation.rotate(qRadiansToDegrees(dy), camera.right);

    QVector4D column = model.column(3);
    model.setColumn(3, {0.0f, 0.0f, 0.0f, 1.0f});
    model = rotation * model;
    model.setColumn(3, column);

    mouse.lastX = x;
    mouse.lastY = y;
  }
}
```

Before applying the rotation, the molecular model is positioned at the center of the scene, zeroing (with the exception of the last element, which is set to one) the fourth column of the model matrix, which defines the translation of the model in world-space.

6.1.2 Model Panning

The panning of the molecular model, which is activated by moving the mouse when the middle button is pressed, takes place in view-space on the plane facing the user.

Applying the translation in view-space rather than in world-space, gives a more natural result for the user. The new position of the model will be the one that the user instinctively associates to the relative movement of the mouse.

Listing 6.2 : mouse move handler - model panning

```
void OpenGLWidget::mouseMoveEvent(QMouseEvent *event)
{
  int x = event->pos().x();
  int y = event->pos().y();
  float factor = 0.005f;

  // model rotating
  // ...

  // model panning
  if (mouse.MouseMiddleButton)
  {
    factor *= 10.0f;

    float dx = factor * (x - mouse.lastX);
    float dy = factor * (mouse.lastY - y);

    QMatrix4x4 translation;
    translation.translate(dx * camera.right);
    translation.translate(dy * camera.up);
    model = translation * model;

    mouse.lastX = x;
    mouse.lastY = y;
```

```
    factor *= 0.1f;
  }

}
```

Note how the translations are applied with respect to the up and right axes of the camera, which together with the front axis define the view-space coordinate system.

6.1.3 Camera Orbiting

By moving the mouse when the right button is pressed, the user can move the camera on a circumference arc while it is still facing the center point of the scene. This means that the camera can not be moved freely, but it can only orbit around the origin of the world-space.

Listing 6.3 : mouse move handler - camera orbiting

```
void OpenGLWidget::mouseMoveEvent(QMouseEvent *event)
{
  int x = event->pos().x();
  int y = event->pos().y();
  float factor = 0.005f;

  // model rotating
  // ...

  // model panning
  // ...

  // camera orbiting
  if (mouse.MouseRightButton)
  {
    float dx = factor * (x - mouse.lastX);
    float dy = factor * (y - mouse.lastY);

    camera.yaw += qRadiansToDegrees(dx);
    camera.pitch = qBound(-MAX_PITCH, camera.pitch + qRadiansToDegrees(dy),
+MAX_PITCH);

    camera.eye = {
      cos(qDegreesToRadians(camera.yaw)) * cos(qDegreesToRadians(camera.pitch)),
      sin(qDegreesToRadians(camera.pitch)),
      sin(qDegreesToRadians(camera.yaw)) * cos(qDegreesToRadians(camera.pitch)),
    };

    camera.eye = camera.radius * camera.eye.normalized();

    camera.front = (camera.center - camera.eye).normalized();
    camera.right = QVector3D::crossProduct(camera.front, {0.0f, 1.0f,
0.0f}).normalized();
    camera.up = QVector3D::crossProduct(camera.right,
camera.front).normalized();

    mouse.lastX = x;
    mouse.lastY = y;
```

```
  }
}
```

eye and center represent respectively the position of the camera and the center of the scene, both in world-space. Note how the camera rotations are applied around the up and right axes. The rotation angle around the right axis (also known as the pitch angle) is limited to values below 90 degrees to avoid to overturn the camera. After the camera's orbiting movement is calculated, the up, right and front axes that define the view-space are orthonormalized using the Gram-Schmidt process, so as to form once again an orthonormal triplet of vectors.

6.1.4 Atom Selection

By hovering the mouse cursor over an atom of the molecular model, the user can retrieve its data, which are displayed in the right panel of the application window.

In order to correctly detect the atom under the mouse cursor, we first add another color buffer output to the geometry step and connect it to a texture we called colour.

The following listing update the previous impostor fragment shader, see Listing 2.3. Here we simply add the new color buffer named colour to the outputs and write it with the value received from the previous shader.

Listing 6.4 : impostor fragment shader

```
#version 450

layout (location = 0) out vec3 center;
layout (location = 1) out vec3 normal;
layout (location = 2) out vec3 albedo;
layout (location = 3) out vec3 colour;

struct Impostor
{
  vec3 center;
  vec3 normal;
};

void SetImpostor(out Impostor impostor);

in Fragment
{
  flat vec3 center;
  flat float radius;
  flat vec3 albedo;
  flat vec3 colour;
  smooth vec2 offset;
} frag;

uniform mat4 projection;
```

```
void main()
{
  Impostor impostor;
  SetImpostor(impostor);

  // frag depth
  // ...

  center = impostor.center;
  normal = impostor.normal;
  albedo = frag.albedo;
  colour = frag.colour;
}
```

colour is calculated in the new impostor vertex shader, which replaces the version in Listing 2.1, and reaches unchanged the fragment shader above. The way we calculate colour is the same as seen in Listing 4.5 for the multi-silhouette colors.

Listing 6.5 : impostor vertex shader

```
#version 450 core

layout (location = 0) in vec3 center;
layout (location = 1) in float radius;
layout (location = 2) in vec3 albedo;
layout (location = 3) in vec2 number;

out Vertex
{
  vec3 center;
  float radius;
  vec3 albedo;
  vec3 colour;
} vert;

void main()
{
  float AtomNumber = number[0];
  uvec3 mask = uvec3(0x00FF0000, 0x0000FF00, 0x000000FF);
  uvec3 shift = uvec3(16, 8, 0);
  vec3 colour = ((uvec3(AtomNumber) & mask) >> shift) / 255.0f;

  vert.center = center;
  vert.radius = radius;
  vert.albedo = albedo;
  vert.colour = colour;
}
```

Therefore the texture colour contains the silhouette of the molecular model with each atom drawn with a unique color calculated based on its serial number.

Let's now go back to the mouse move handler to see how atom selection is implemented.

Listing 6.6 : mouse move handler - atom selection

```
void OpenGLWidget::mouseMoveEvent(QMouseEvent *event)
{
  int x = event->pos().x();
  int y = event->pos().y();
  float factor = 0.005f;

  // model rotating
  // ...

  // model panning
  // ...

  // camera orbiting
  // ...

  // atom selection
  makeCurrent();
  glBindFramebuffer(GL_READ_FRAMEBUFFER, FBO::GEOMETRY);
  {
    int h = geometry().height();

    glReadBuffer(GL_COLOR_ATTACHMENT3);

    QVector<GLubyte> pixel(3);
    glReadPixels(x, h - y, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &(pixel[0]));

    QVector<unsigned int> shift = {16, 8, 0};

    int AtomNumber = 0;
    for (int i = 0; i < pixel.size(); i++)
    {
      AtomNumber |= pixel[i] << shift[i];
    }

    if (trajectory.atoms.keys().contains(AtomNumber))
    {
      emit MouseHoverSignal(trajectory.atoms[AtomNumber]);
    }
    else
    {
      emit MouseNotHoverSignal();
    }
  }
  glBindFramebuffer(GL_FRAMEBUFFER, defaultFramebufferObject());
  doneCurrent();
}
```

We set the geometry FBO as the read framebuffer and we specify the fourth color buffer (where we previously wrote colour) as the source for the subsequent call to glReadPixels, which gives us the color of the pixel under the mouse cursor.

We convert the color read into a number (by reversing the sequence of bitwise operations we did in the vertex shader) and check if it matches the serial number of an atom. Finally, we emit the proper signal in order to update the right panel of the application window.

6.1.5 Camera Zooming

By scrolling the mouse wheel, the user can move the camera closer or further away from the center of the scene, changing the orbiting radius of the camera around it.

Listing 6.7 : mouse wheel scroll handler

```
void OpenGLWidget::wheelEvent(QWheelEvent *event)
{
  if (mouse.MouseLeftButton || mouse.MouseMiddleButton || mouse.MouseRightButton
|| mouse.MouseWheel) return;

  float dy = event->angleDelta().y() * 0.125f / 15;
  camera.radius = qBound(MIN_RADIUS, camera.radius + dy, MAX_RADIUS);
  camera.eye = camera.radius * camera.eye.normalized();
}
```

eye represents the position of the camera in world-space and radius represents the orbiting radius of the camera around the center of the scene. The orbiting radius is limited to positive values to prevent the camera from overturning.

6.2 Right Panel

This widget is made up of different parts, some of which return information on the molecular simulation and others allow to configure the scene.

We will review each element of the panel, starting from the top and going down.

6.2.1 Atom and Residue Data Tab Widget

This tab widget shows the data of the atom and residue selected on mouseover.

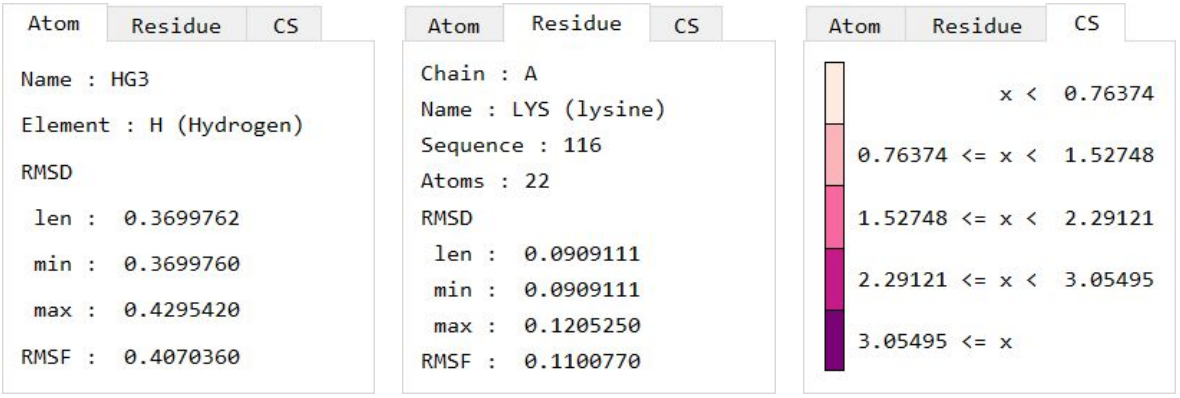


Figure 6.2 : atom and residue data tab widget

The first two tabs show the data of the atom and residue selected, while the third tab shows the outline color scheme for the selected entity with the values range of each color step.

6.2.2 Trajectory Data Widget

This widget shows the minimum and maximum values of RMSD and RMSF calculated on the entire trajectory for both atoms and residues.

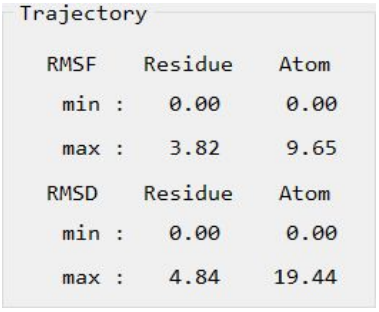


Figure 6.3 : trajectory data widget

These values are clearly the same for each atom and residue and are also constant during the all molecular simulation.

6.2.3 Object Outlining Widget

This widget gives the user the ability to enable or disable the object outlining feature and to modify its parameters.

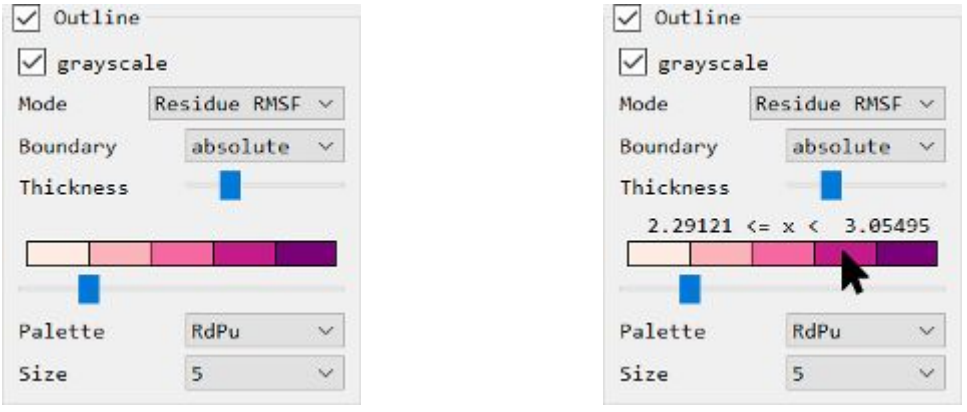


Figure 6.4 : object outlining widget

The object outlining widget has several elements.

- **outline checkbox** : if checked, enables object outlining

- **grayscale checkbox** : if checked, albedo values of the molecular model are converted in grayscale values

- **mode combobox** : sets the quantity that the outline represents, where the possible values are residue RMSF, residue RMSD, atom RMSF and atom RMSD

- **boundary combobox** : the minimum and maximum values on which the outline color scheme is constructed can be absolute, based on the minimum and maximum values of the entire simulation, or relative, based on the minimum and maximum values of each atom or residue

- **thickness slider** : sets the thickness of the outline

- **color scheme range values** : hovering the mouse cursor, shows the values range of the color step selected (see Figure 6.4, left side)

- **outline filter slider** : filters the outline based on the value of the represented quantity, atoms or residues with a lower value than the slider are not outlined

- **palette combobox** : sets the color palette of the outline color scheme

- **size combobox** : sets the size of the outline color scheme

6.2.4 Lighting Tab Widget

This widget gives the user the ability to tweak and experiment with the parameters of the lighting model.

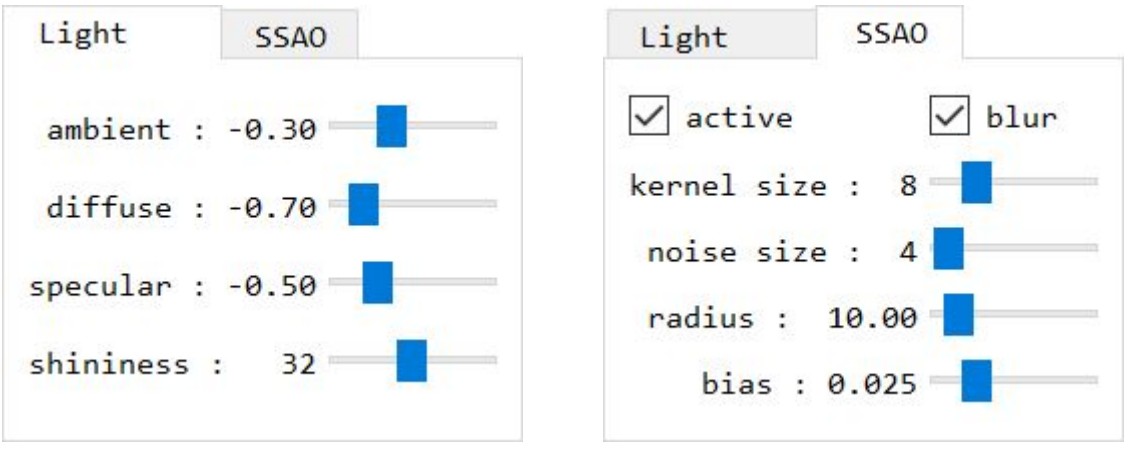


Figure 6.5 : lighting tab widget

In the light tab, the user can tweak the light properties to adjust the lighting components of the blinn-phong lighting model described in Section 3.1.

In the SSAO tab, the user can enable or disable screen-space ambient occlusion, along with the blur effect, and tweak its parameters to find the right balance between quality and performance.

6.3 Bottom Bar

The bottom bar contains the playback widget that allows the user to move between the steps of the molecular simulation.

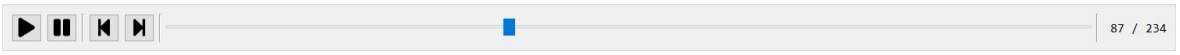


Figure 6.6 : playback widget

By pressing the play button, the simulation is executed automatically and cyclically until the pause button is pressed, which interrupts the playback at the current step.

The step-backward and step-forward buttons allow the user to move one step at a time back and forth in the simulation.

Moreover, a desired step can be selected directly by sliding the slider to the corresponding position.

6.4 Shneiderman’s Mantra

With Shneiderman's Mantra in mind, that is "overview first, zoom and filter, then details-on-demand", we have imagined the following usage scenario for our application.

1. The user can use the residue RMSF to identify the most flexible parts of the protein.

2. Then the user can use the residue RMSD to identify the steps of the simulation where this residues have the maximum deviation from their reference conformation.

3. Finally the user can use the atom RMSF and RMSD to understand which are the atoms that cause the deviation in the residues.

7 Conclusions

Making informed decisions in the drug design process requires estimating the interaction energy between the protein and the ligand and understanding which parts of the molecule influence their binding.

In nowadays molecular visualization packages the methods available to visualize the interaction forces that drive molecular simulations are limited and most of times they consist in simple 2D plots, making difficult to understand the real 3D arrangement of the molecular conformations.

7.1 Addressed Problem and Contribution

The goal we set at the beginning of this dissertation, is to display directly on the 3D models of a molecular simulation the information useful in understanding the interactions between protein and ligand. We decided to concentrate our efforts on two important quantities in the field of molecular simulations and drug design: the root mean square deviation (or RMSD) and the root mean square fluctuation (or RMSF) of both atoms and residues.

The approach we proposed to visualize RMSD and RMSF directly on the molecule's 3D model is to outline the parts of the molecule involved into the study (which can be both individual atoms and residues) and map the values of RMSD and RMSF on a color scheme, which is then applied to the outline.

7.2 Accomplished Results

Our goal, as stated above, can be considered achieved.

We have implemented a real-time molecular simulation viewer that allows the user to move between the steps of a simulation and interact with its molecular 3D models.

With nowadays molecular visualization packages, a user interested in the RMSD and RMSF values of a simulation, needs typically to consult a 2D plot that is usually shown in a dedicated window. The chart takes up space on the screen and distracts the user from the 3D simulation.

On the contrary, our application allows the user to view the RMSD and RMSF values directly on the 3D models of the simulation, without having to consult a chart or a table with the values.

This feature gives the user the ability to recognize which are the most mobile or flexible parts of a molecule, both at the atomic and residue level, directly by visualizing and interacting with the 3D models of the simulation.

Moreover, by observing how the values of RMSD and RMSF vary during the simulation, the user can understand or formulate hypotheses about the interactions between the molecules (or between parts of them).

7.3 Application Improvements

The work presented in this report shows how a quantity that is typically represented with a dedicated 2D plot can be inserted and displayed in another context, allowing to aggregate different classes of information in a single image (in our case the spatial information given from the 3D model and the values of RMSD and RMSF), and at the same time preserving an easy understanding of each information class.

We hope that our work will inspire future research in the field of molecular visualization and more generally in scientific and non-scientific visualization.

We conclude this report with a list of possible ideas to improve and expand our application, many of which have remained out of the version presented here for timing reasons.

- We calculated RMSD and RMSF taking as reference conformation the first step of the simulation. It would be interesting to compare the values obtained with those calculated from other reference conformations, such as the average conformation of the trajectory or the native conformation of a protein.

- The RMSD for a single atom expresses the distance (in angstroms) from its reference position in its residue. Since for a conformation of a single element $N = 1$, the Equation 5.1 can be simplified in the following equation.

$$RMSD = \left\| x_{1,\,t} - x_{1,\,t_{ref}} \right\|$$

  This quantity represents the length of the atom displacement vector relative to its reference position. If we normalize this vector, we get the direction of the displacement.

  It would be interesting and graphically challenged to also display this information directly on a 3D molecular model. An idea could be to use a 2D or 3D glyph, for example in the shape of an arrow, positioned near the atom.

- It is mandatory to give the user the ability to "access" the hidden parts of a molecule, that is those atoms and residues occluded from other parts of the molecule itself or from other molecules.

- It would be useful to add another outline filter based on the distance of atoms and residues from the ligand, to recognize its interactions with the protein.

- It would definitely be better to show the atoms and residues information in a pop-up box next to the mouse cursor, making sure not to hide the selected object.

Bibliography

[01] Pedro Hermosilla Casajús. *Advanced Inspection Techniques for Molecular Simulations.* 2017.

[02] P. Vázquez, P. Hermosilla, V. Guallar, J. Estrada, A. Vinacua. *Visual Analysis of Protein-Ligand Interactions.* 2018.

[03] René Fester Kratz. *Molecular & Cell Biology For Dummies.* 2009.

[04] Rajendra Kumar Singh, A.S. Ethayathulla, Talat Jabeen, Sujata Sharma, Punit Kaur & Tej P. Singh. *Aspirin induces its anti-inflammatory effects through its specific binding to phospholipase A2: Crystal structure of the complex formed between phospholipase A2 and aspirin at 1.9 Å resolution.* 2005.

[05] John Kessenich, Graham Sellers, Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL (ninth edition).* 2016.

[06] Kouichi Matsuda, Rodger Lea. *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL.* 2013.

[07] Joey de Vries. *Learn OpenGL.* 2017.

[08] Jason L. McKesson. *Learning Modern 3D Graphics Programming.* 2012.

[09] Eric Haines, Naty Hoffman, Tomas Möller. *Real-Time Rendering (fourth edition).* 2018.

[10] Guillaume Lazar, Robin Penea. Mastering Qt 5. 2016.